Scientific Research Publishing

Applied Mathematics, 2021, 12, 900-918
https://www.scirp.org/journal/am
ISSN Online: 2152-7393
ISSN Print: 2152-7385

# A Novel Method for Solving Ordinary Differential Equations with Artificial Neural Networks

**Roseline N. Okereke, Olaniyi S. Maliki, Ben I. Oruh**

Department of Mathematics, Michael Okpara University of Agriculture, Umudike, Nigeria
Email: okerekern@gmail.com, somaliki@gmail.com, oruhben@gmail.com

**How to cite this paper:** Okereke, R.N., Maliki, O.S. and Oruh, B.I. (2021) A Novel Method for Solving Ordinary Differential Equations with Artificial Neural Networks. *Applied Mathematics*, **12**, 900-918.
https://doi.org/10.4236/am.2021.1210059

**Received:** November 19, 2020
**Accepted:** October 18, 2021
**Published:** October 21, 2021



## Abstract

This research work investigates the use of Artificial Neural Network (ANN) based on models for solving first and second order linear constant coefficient ordinary differential equations with initial conditions. In particular, we employ a feed-forward Multilayer Perceptron Neural Network (MLPNN), but bypass the standard back-propagation algorithm for updating the intrinsic weights. A trial solution of the differential equation is written as a sum of two parts. The first part satisfies the initial or boundary conditions and contains no adjustable parameters. The second part involves a feed-forward neural network to be trained to satisfy the differential equation. Numerous works have appeared in recent times regarding the solution of differential equations using ANN, however majority of these employed a single hidden layer perceptron model, incorporating a back-propagation algorithm for weight updation. For the homogeneous case, we assume a solution in exponential form and compute a polynomial approximation using statistical regression. From here we pick the unknown coefficients as the weights from input layer to hidden layer of the associated neural network trial solution. To get the weights from hidden layer to the output layer, we form algebraic equations incorporating the default sign of the differential equations. We then apply the Gaussian Radial Basis function (GRBF) approximation model to achieve our objective. The weights obtained in this manner need not be adjusted. We proceed to develop a Neural Network algorithm using MathCAD software, which enables us to slightly adjust the intrinsic biases. We compare the convergence and the accuracy of our results with analytic solutions, as well as well-known numerical methods and obtain satisfactory results for our example ODE problems.

## Keywords

Ordinary Differential Equations, Multilayer Perceptron Neural Networks,

Gaussian Radial Basis Function, Network Training, MathCAD (Computer Aided Design) 14, IBM-SPSS (Statistical Package for Social Science) 23

## 1. Introduction

The beginning of Neuro-computing is often taken to be the research article of McCulloch and Pitts [1] published in 1943, which showed that even simple types of neural networks could, in principle, compute any arithmetic or logical function, was widely read and had great influence. Other researchers, principally von Neumann, wrote a book [2] in which the suggestion was made that the research into the design of brain-like or brain-inspired computers might be of great interest and benefit to scientific and technological knowledge.

We present a new perspective for obtaining solutions of initial value problems using Artificial Neural Networks (ANN). We discover that neural network based model for the solution of ordinary differential equations (ODE) provides a number of advantages over standard numerical methods. Firstly, the neural network based solution is differentiable and is in closed analytic form. On the other hand most other techniques offer a discretized solution or a solution with limited differentiability. Secondly, the neural network based method for solving a differential equation provides a solution with very good generalization properties. The major advantage here is that our method reduces considerably the computational complexity involved in weight updating, while maintaining satisfactory accuracy.

### Neural Network Structure

A neural network is an inter-connection of processing elements, units or nodes, whose functionality resemble that of the human neurons [3]. The processing ability of the network is stored in the connection strengths, simply called weights, which can be obtained by a process of adaptation to, a set of training patterns. Neural network methods can solve both ordinary and partial differential equations. Furthermore, it relies on the function approximation property of feed forward neural networks which results in a solution written in a closed analytic form. This form employs a feed forward neural network as a basic approximation element [4] [5]. Training of the neural network can be done either by any optimization technique which in turn requires the computation of the gradient the error with respect to the network parameters, by regression based model or by basis function approximation. In any of these methods, a trial solution of the differential equation is written as a sum of two parts, proposed by Lagaris [6]. The first part satisfies the initial or boundary conditions and contains no adjustable parameters. The second part contains some adjustable parameters that involves feed forward neural network and is constructed in a way that does not affect the initial or boundary conditions. Through the construction, the trial so-

lution, initial or boundary conditions are satisfied and the network is trained to satisfy the differential equation. The general flowchart for neural network training (or learning) is given below in Figure 1.

## 2. Neural Networks as Universal Approximators

Artificial neural networks can make a nonlinear mapping from the inputs to the outputs of the corresponding system of neurons, which is suitable for analyzing the problem defined by initial/boundary value problems that have no analytical solutions or which cannot be easily computed. One of the applications of the multilayer feed forward neural network is the global approximation of real valued multivariable function in a closed analytic form. Namely such neural networks are universal approximators. It is discovered in the literature that multilayer feed forward neural networks with one hidden layer using arbitrary squashing functions are capable of approximating any Borel measurable function from one finite dimensional space to another with any desired degree of accuracy. This is made clear in the following theorem.

### Universal Approximation Theorem

The universal approximation theorem for MLP was proved by Cybenko [7] and Hornik *et al.* [8] in 1989. Let $\boldsymbol{I}_n$ represent an $n$-dimensional unit cube containing all possible input samples $\boldsymbol{x}=(x_1,x_2,\cdots,x_n)$ with $x_i\in[0,1]$, $i=1,2,\cdots,n$. Let $C(\boldsymbol{I}_n)$ be the space of continuous functions on $\boldsymbol{I}_n$, given a continuous sigmoid function $\varphi(\cdot)$, then the universal approximation theorem states that the finite sums of the form

$$y_k = y_k(\boldsymbol{x},\boldsymbol{w}) = \sum_{i=1}^{N_2} w_{ki}^3 \varphi\left(\sum_{j=0}^{n} w_{ki}^2 x_j\right),\ k=1,2,\cdots,m \tag{1}$$

are dense in $C(\boldsymbol{I}_n)$. This simply means that given any function $f\in C(\boldsymbol{I}_n)$ and $\varepsilon>0$, there is a sum $y(\boldsymbol{x},\boldsymbol{w})$ of the above form that satisfies

$$\left|y(\boldsymbol{x},\boldsymbol{w})-f(\boldsymbol{x})\right|<\varepsilon,\ \forall\boldsymbol{x}\in\boldsymbol{I}_n. \tag{2}$$

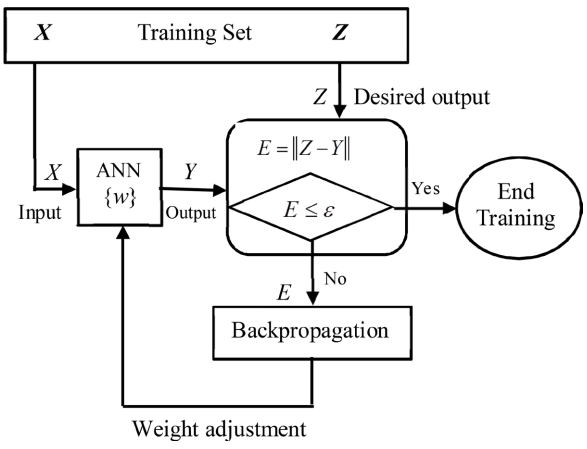


Figure 1. Network training flowchart.

## 3. Gradient Computation

Minimization of error function can also be considered as a procedure for training the neural network [9], where the error corresponding to each input vector $\boldsymbol{x}$ is the value $f(\boldsymbol{x})$ which has to become zero. In computing this error value, we require the network output as well as the derivatives of the output with respect to the input vectors. Therefore, while computing error with respect to the network parameters, we need to compute not only the gradient of the network but also the gradient of the network derivatives with respect to its inputs. This process can be quite tedious computationally, and we briefly outline this in what follows.

### 3.1. Gradient Computation with Respect to Network Inputs

Next step is to compute the gradient with respect to input vectors, for this purpose let us consider a multilayer perceptron (MLP) neural network with $n$ input units, a hidden layer with $m$ sigmoid units and a linear output unit. For a given input vector $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ the network output is written:

$$N(\boldsymbol{x}, \boldsymbol{p}) = \sum_{i=1}^{m} v_j \varphi(z_j), \quad z_j = \sum_{i=1}^{n} w_{ji} x_i + u_j \tag{3}$$

$w_{ji}$ denotes the weight from input unit $i$ to the hidden unit $j$, $v_j$ denotes weight from the hidden unit $j$ to the output unit, $u_j$ denotes the biases, and $\varphi(z_j)$ is the sigmoid activation function.

Now the derivative of networks output $N$ with respect to input vector $x_i$ is:

$$\frac{\partial}{\partial x_i} N(\boldsymbol{x}, \boldsymbol{p}) = \frac{\partial}{\partial x_i}\left(\sum_{j=1}^{m} v_j \varphi(z_j)\right) = \sum_{j=1}^{m} v_j w_{ji} \varphi^{(1)} \tag{4}$$

where $\varphi^{(1)} \equiv \partial\varphi(\boldsymbol{x})/\partial\boldsymbol{x}$. Similarly, the $k^{\text{th}}$ derivative of $N$ is computed as;

$$\partial^k N / \partial x_i^k = \sum_{j=1}^{m} v_j w_{ji}^k \varphi_j^{(k)}$$

Where $\varphi_j \equiv \varphi(z_j)$ and $\varphi^{(k)}$ denotes the $k^{\text{th}}$ order derivative of the sigmoid activation function.

### 3.2. Gradient Computation with Respect to Network Parameters

Network's derivative with respect to any of its inputs is equivalent to a feed-forward neural network $N_k(\boldsymbol{x})$ with one hidden layer, having the same values for the weights $w_{ji}$ and thresholds $u_j$ and with each weight $v_j$ being replaced with $v_j p_j$. Moreover, the transfer function of each hidden unit is replaced with the $k^{\text{th}}$ order derivative of the sigmoid function. Therefore, the gradient of $N_k$ with respect to the parameters of the original network can easily obtained as:

### 3.3. Network Parameter Updation

After computation of derivative of the error with respect to the network parameter has been defined then the network parameters $v_j$, $u_j$ and $w_{ji}$ upda-

tion rule is given as,

$$v_j(t+1)=v_j(t)+\mu\frac{\partial N_k}{\partial v_j},\quad u_j(t+1)=u_j(t)+\eta\frac{\partial N_k}{\partial u_j},\quad w_{ji}(t+1)=w_{ji}(t)+\gamma\frac{\partial N_k}{\partial w_{ji}} \tag{5}$$

where $\mu$, $\eta$ and $\gamma$ are the learning rates, $i=1,2,\cdots,n$ and $j=1,2,\cdots,m$.

Once a derivative of the error with respect to the network parameters has been defined it is then straightforward to employ any optimization technique to minimize error function.

## 4. General Formulation for Differential Equations

Let us consider the following general differential equations which represent both ordinary and partial differential equations Majidzadeh [10]:

$$G\left(x,\psi(x),\nabla\psi(x),\nabla^2\psi(x),\cdots\right)=0,\ \forall x\in D, \tag{6}$$

subject to some initial or boundary conditions, where $x=(x_1,x_2,\cdots,x_n)\in\mathbb{R}^n$, $D\subset\mathbb{R}^n$ denotes the domain, and $\psi(x)$ is the unknown scalar-valued solution to be computed. Here, $G$ is the function which defines the structure of the differential equation and $\nabla$ is a differential operator. Let $\psi_t(x,p)$ denote the trail solution with parameters (weights, biases) $p$. Tian Qi *et al.* [11], gave the following as the general formulation for the solution of differential equations (5) using ANN. Now, $\psi_t(x,p)$ may be written as the sum of two terms

$$\psi_t(x,p)=A(x)+F\left(x,N(x,p)\right), \tag{7}$$

where $A(x)$ satisfies initial or boundary condition and contains no adjustable parameters, whereas $N(x,p)$ is the output of feed forward neural network with the parameters $p$ and input data $x$. The function $F\left(x,N(x,p)\right)$ is actually the operational model of the neural network. Feed forward neural network (FFNN) converts differential equation problem to function approximation problem. The neural network $N(x,p)$ is given by;

$$N(x,p)=\sum_{j=1}^{m}v_j\sigma(z_j),\quad z_j=\sum_{i=1}^{n}w_{ji}x_i+u_j \tag{8}$$

$w_{ji}$ denotes the weight from input unit $i$ to the hidden unit $j$, $v_j$ denotes weight from the hidden unit $j$ to the output unit, $u_j$ denotes the biases, and $\sigma(z_j)$ is the sigmoid activation function.

### Neural Network Training

The neural network weights determine the closeness of predicted outcome to the desired outcome. If the neural network weights are not able to make the correct prediction, then only the biases need to be adjusted. The basis function we shall apply in this work in training the neural network is the sigmoid activation function given by

$$\sigma(z_j)=\left(1+\mathrm{e}^{-z_j}\right)^{-1}. \tag{9}$$

## 5. Method for Solving First Order Ordinary Differential Equation

Let us consider the first order ordinary differential equation below

$$\psi'(x) = f(x,\psi),\ x \in [a,b] \tag{10}$$

with initial condition $\psi(a) = A$. In this case, the ANN trial solution may be written as

$$\psi_t(x,p) = A + xN(x,p), \tag{11}$$

where $N(x,p)$ is the neural output of the feed forward network with one input data $x$ with parameters $p$. The trial solution $\psi_t(x,p)$ satisfies the initial condition. Now let us consider a first order differential equation:

$$\psi'(x) - \psi = 0,\ x \in [0,1],\ \psi(0) = 1 \tag{12}$$

with trial solution:

$$\psi_t(x,p) = A + xN(x,p), \tag{13}$$

where $x$ is the input to neural network model and $p$ represents the parameters—weights and biases.

$$w_t(0,p) = A + (0)N(0,p) = A = 1 \ \Rightarrow \psi_t(x,p) = 1 + xN(x,p), \tag{14}$$

To solve this problem using neural network, we shall employ a neural network architecture with three layers. One input layer with one neuron; one hidden layer with three neurons and one output layer with one output unit, as depicted below in Figure 2.

Each neuron is connected to other neurons of the previous layer through adaptable synaptic weights $w_j$ and biases $u_j$. Now,

$\psi_t(x_i,p) = 1 + x_i N(x_i,p)$, with $N(x_i,p) = \sum_{j=1}^{3} v_j \sigma(z_j)$, and

$$\sum_{j=i}^{3} v_j \sigma(z_j) = v_1\sigma(z_1) + v_2\sigma(z_2) + v_3\sigma(z_3), \tag{15}$$

where $z_1 = x_1 w_{11} + u_1,\ z_2 = x_1 w_{12} + u_2,\ z_3 = x_1 w_{13} + u_3$.

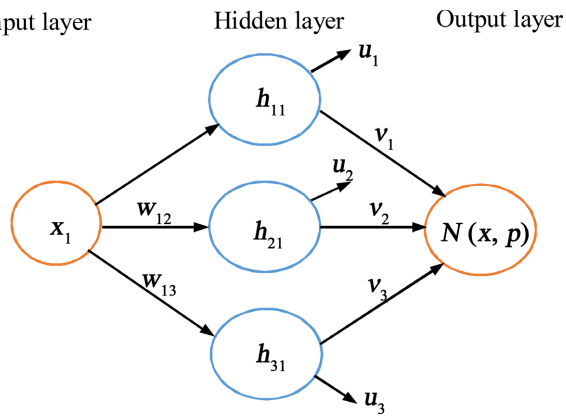


Figure 2. Schematic for $N(x,p)$.

Now, in solving ordinary differential equations, we assume a solution to the homogeneous part and approximate the function using SPSS model, which estimates the regression coefficients in the multiple regression mode. These coefficients are what we use as the weights from the input layer to the hidden layer. The condition placed on $y_a(x) = f(x)$, say, where $y_a(x)$ is the assumed solution is that $f(x) \neq 0$.

Any exponential function, $y(x) = \mathrm{e}^{\alpha x}$, where $\alpha \in \mathbb{R}$, is a part of solution to any first order ordinary differential equation. We regress that function using excel spreadsheet and SPSS model as follows:

Assuming we let $y_a(x) = f(x) = \mathrm{e}^{2x}, x \in [0,1]$, be a solution to a given first order ordinary differential equation, $y' - 2y = f(x)$, defined on the given interval. Then dividing the interval into 10 equidistant points, we use excel spreadsheet to find values of $y_a(x)$ at all the points shown in Table 1, then use SPSS to regress and get the weights which we designate weights from input layer to hidden layer.

The above is followed by the display of the SPSS 20 output of the data in Table 2, from which we pick the weights.

Looking at Table 2, we see that the cubic curve fits perfectly the assumed solution. Therefore, we pick the coefficients: 2.413, 0.115 and 3.860 as the weights from input layer to the hidden layer.

The next task is to obtain the weights from hidden layer to the output layer. We shall find $f(x)$, a real function of a real valued vector $x = (x_1, x_2, \cdots, x_d)^{\mathrm{T}}$ and a set of functions, $\{\varphi_i(x)\}$ called the elementary functions such that

$$\hat{f}(x, v) = \sum_{i=1}^{N} v_i \varphi_i(x) \tag{16}$$

is satisfied, where $v_i$ are real valued constants such that $\left| f(x) - \hat{f}(x, v) \right| < \varepsilon$. When one can find coefficients $v_i$ that make $\varepsilon$ arbitrarily small for any function $f(.)$ over the domain of interest, we say that the elementary function set $\{\varphi_i(.)\}$ has the property of universal approximation over the class of functions $f(.)$. There are many possible elementary functions we can choose from which we will show later. Now if the number of input vectors $x_i$ is made equal to the

**Table 1.** Table of values for the relationship between $y_a$ and $x$.

| $X$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $Y$ | 1 | 1.2214 | 1.49183 | 1.82212 | 2.22554 | 2.71828 | 3.32012 | 4.0552 | 4.95303 | 6.04965 | 7.38906 |

**Table 2.** Model summary and parameter estimates.

| Equation | $R^2$ | Constant | $b_1$ | $b_2$ | $b_3$ |
|---|---|---|---|---|---|
| Linear | 0.930 | 0.240 | 6.109 | | |
| Quadratic | 0.998 | 1.126 | 0.205 | 5.904 | |
| Cubic | 1.000 | 0.987 | 2.413 | 0.115 | 3.860 |

The Independent variable is $X$

number of elementary functions $\varphi_i(.)$, then the normal equations can be given as;

$$\begin{bmatrix} \varphi_1(x_1) & \cdots & \varphi_N(x_1) \\ \vdots & \ddots & \vdots \\ \varphi_1(x_N) & \cdots & \varphi_N(x_N) \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_N) \end{bmatrix} \tag{17}$$

and the solution becomes $v = \phi^{-1} f$, where $v$ becomes a vector with the coefficients, $f$ is a vector composed of the values of the function at the $N$ points, and $\phi$ the matrix with entries given by values of the elementary functions at each of the $N$ points in the domain. An important condition that must be placed in the elementary functions is that the inverse of $\phi$ must exist. In general, there are many sets $\{\varphi_i(.)\}$ with the property of universal approximation for a class of functions. We would prefer a set $\{\varphi_i(.)\}$ over another $\{\gamma_i(.)\}$ if $\{\varphi_i(.)\}$ provides a smaller error $\varepsilon$ for a pre-set value of *N*. This means that the speed of convergence of the approximation is also an important factor in the selection of the basis. As we mentioned earlier, there are many possible elementary basis functions we can choose from. A typical example of basis function is the Gaussian basis specified by:

$$G(z) = G(\|x - c_i\|) = \exp(-z^2); i = 1, 2, \cdots, N$$

However in neuro-computing, the most popular choice for elementary functions is the radial basis function (RBFs) where $\varphi_i(x)$ is given;

$$\varphi_i(x) = \exp\left(-\frac{|x - x_i|^2}{2\sigma^2}\right) \tag{18}$$

where $\sigma^2$ is the variance of input vectors *x*. It is this last basis function that we shall adopt in generating our weights from hidden layer to output layer. At this point we shall divide a given interval into a certain number of points equidistant from each other and choose input vectors $x_i$ to conform with the number of elementary functions $\varphi_i(.)$ to make the basis function implementable. Here $N = 3$ and the solution $v = \phi^{-1} f$ is given by;

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \varphi_3(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \varphi_3(x_2) \\ \varphi_1(x_3) & \varphi_2(x_3) & \varphi_3(x_3) \end{bmatrix}^{-1} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{19}$$

where $v, \phi, f$ are as defined before, and

$$\varphi_i(x) = \exp\left(-\frac{|x - x_i|^2}{2\sigma^2}\right), \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2, \quad \bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{20}$$

Now to compute $f(x)$ depends on the nature of a given differential equation. For first, second or higher order homogeneous ordinary differential equations, we form linear, quadratic or higher order polynomial equations incorporating the default signs of the terms in the differential equations. For non-homogeneous ordinary differential equations, we use the forcing functions. When

the weights of the neural network are obtained by these systematic ways, there is no need to adjust all the parameters in the network, as postulated by previous researchers, in order to achieve convergence. All that is required is a little adjustment of the biases, and these are fixed to lie in a given interval and convergence to a solution with an acceptable minimum error is achieved. When the problem of obtaining the parameters is settled, it becomes easy to solve any first, second or higher order ordinary differential equation using the neural network model appropriate to it. We shall restrict this study to first and second order linear ordinary homogeneous differential equations. To compute the prediction error we use the squared error function defined as follows:

$$E = 0.5\left(\psi_d - \psi_p\right)^2 \tag{21}$$

where $\psi_d$ represents the desired output and $\psi_p$ the predicted output. The same procedure can be applied to second and third order ODE. For the second order initial value problem (IVP):

$$\psi''(x) = f\left(x, \psi, \psi'(x)\right),\ \psi(0) = A,\ \psi'(0) = B \tag{22}$$

The trial solution is written

$$\psi_t(x) = A + Bx + x^2 N(x, p) \tag{23}$$

where $A, B \in \mathbb{R}$ and $N(x_i, p) = \sum_{j=1}^{3} v_j \sigma\left(z_j\right)$, and for two point BC:

$\psi(0) = A, \psi(1) = B$ the trial solution in this case is written:

$\psi_t(x) = A(1-x) + Bx + x(1-x)N(x, p)$.

Now, we first demonstrate the computational complexity involved in adjusting all parameters in order to update the weights and getting a close approximation to the desired result. Subsequently, we proceed to our main results and analysis that displays the ease of computation achieved by our novel method of adjusting only the biases. The former adjustment or weight updation is done using the backpropagation algorithm. Therefore, we need to train the network so we can apply the backpropagation algorithm. The basis function we shall apply in this work in training the neural network is the sigmoid activation function given by Equation (9). In a simple neural model as depicted in **Figure 3**, where there is no hidden layer and $x = (x_1, x_2), w = (w_1, w_2)$, we shall assume some figures for illustration. Let the training data be given as $x = (x_1, x_2) = (0.1, 0.2)$; desired output $y_d = 0.02$; initial weights $w = (w_1, w_2) = (0.4, 0.1)$; the bias $b = 1.78$ and predicted output $y_p$. The diagram below shows the neural network training model for the sample data we are considering. Now we proceed to train the network to get the predicted output.

### 5.1. Network Training

First we compute $z = x_1 \cdot w_1 + x_2 \cdot w_2 + b$, which is sum of products and bias, *i.e.*;

$$z = x_1(w_1) + x_2(w_2) + b = 0.1(0.4) + 0.2(0.1) + 1.78 = 1.84 \tag{24}$$

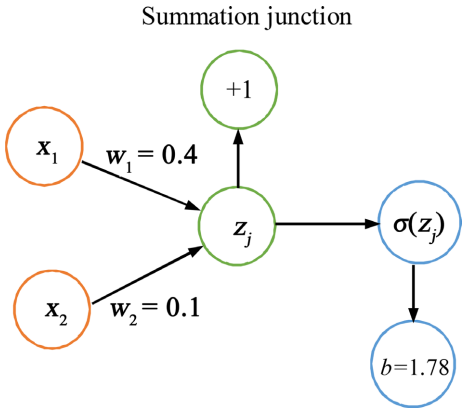


**Figure 3.** Schematic for $N(x, p)$.

Next we apply $z$, as input to the activation function, which in this case is the sigmoid activation function:

$\sigma(z) = \left(1+\mathrm{e}^{-z}\right)^{-1} = \left(1+\mathrm{e}^{-1.84}\right)^{-1} = 0.863$. Hence the predicted output $y_p = 0.863$.

We have seen that the predicted output does not correspond to the desired output, therefore we have to train the network to reduce the prediction error. To compute the prediction error, we use the squared error (21) function defined as follows. Considering the predicted output we calculated above, the prediction error is:

$$E = 0.5(0.02 - 0.863)^2 = 0.355. \tag{25}$$

We observe that the prediction error is huge, so we must attempt to minimize it. We noted previously that the weights determine how close a predicted output is to the desired output. Therefore to minimize the error we have to adjust the weights. This can be achieved using the formula;

$$w_n = w_o + \eta\left(y_d - y_p\right)x \tag{26}$$

where $w_n$ and $w_o$ represent new and old weights respectively. We update the weights using the following:

$w_o$: *current weight* (1.78, 0.4, 0.1);

$\eta$: *network learning rate* = 0.01;

$y_d$: *desired output* = 0.02;

$x$: *current input vector* = (+1, 0.1, 0.2).

$$\therefore\ w_n = [1.78, 0.4, 0.1] + 0.01[0.02 - 0.863][+1, 0.1, 0.2] = [1.772, 0.399, 0.098].$$

With this information we adjust the model and retrain the neural network to get;

$$z = 0.1(0.399) + 0.2(0.098) + 1.772 = 1.79 \ \Rightarrow\ \sigma(z) = \left(1+\mathrm{e}^{-1.79}\right)^{-1} = 0.857$$

$$\therefore\ E = 0.5(0.02 - 0.857)^2 = 0.350.$$

### 5.2. Computation of the Gradient

The error computation not only involves the outputs but also the derivatives of the network output with respect to its inputs. So, it requires computing the gradient of the network derivatives with respect to its inputs. Let us now consider a multilayered perceptron with one input node, a hidden layer with $m$ nodes, and one output unit. For the given inputs $x=\left(x_1, x_2, \cdots, x_n\right)$, the output is given by

$$N\left(x, p\right)=\sum_{j=1}^{m} v_j \sigma\left(z_j\right) \tag{27}$$

where $z_j=\sum_{i=1}^{n} w_{ji} x_i+u_j$, $w_{ji}$ denotes the weight from input unit $i$ to the hidden unit $j$, $v_j$ denotes weight from the hidden unit $j$ to the output unit, $u_j$ denotes the biases, and $\sigma\left(z_j\right)$ is the sigmoid activation function. The derivatives of $N\left(x, p\right)$ with respect to input $x_i$ is

$$\frac{\partial^k N}{\partial x_i^k}=\sum_{j=1}^{m} v_j w_{ji}^k \sigma_j^{(k)}, \tag{28}$$

where $\sigma=\sigma\left(z_j\right)$ and $\sigma^{(k)}$ denotes the $k^{\text{th}}$ order derivative of sigmoid function.

Let $N_\theta$ denote the derivative of the network with respect to its inputs and then we have the following relation

$$N_\theta=D^n N=\sum v_i p_i \sigma_i^{(n)} ; \quad p_j=\prod_{k=1}^{n} w_{jk}^{\lambda^k}, \quad k=\sum_{i=1}^{n} \lambda_i \tag{29}$$

The derivative of $N_\theta$ with respect to other parameters may be obtained as

$$\frac{\partial N_\theta}{\partial v_j}=p_j \sigma_j^{(k)}, \quad \frac{\partial N_\theta}{\partial v_j}=v_j p_j \sigma_j^{(k+1)},$$

$$\frac{\partial N_\theta}{\partial w_{ji}}=x_j v_j p_j \sigma_j^{(k+1)}+v_j \lambda_i w_{ji}^{\lambda_i-1}\left(\prod_{k=1, k \neq 1} w_{ji}^{\lambda_k}\right) \sigma_j^{(k)} \tag{30}$$

Now after getting all the derivatives we can find out the gradient of error. Using general learning method for supervised training we can minimize the error to the desired accuracy. We illustrate the above using the first order ordinary differential equation below

$$\psi'\left(x\right)=f\left(x, \psi\right), \; x \in\left[a, b\right] \tag{31}$$

with initial condition $\psi\left(a\right)=A$. In this case, the ANN trial solution may be written as

$$\psi_t\left(x, p\right)=A+xN\left(x, p\right), \tag{32}$$

where $N\left(x, p\right)$ is the neural output of the feed forward network with one input data $x$ with parameters $p$. The trial solution $\psi_t\left(x, p\right)$ satisfies the initial condition. We differentiate the trial solution $\psi_t\left(x, p\right)$ to get

$$\frac{\mathrm{d}\psi_t\left(x, p\right)}{\mathrm{d}x}=N\left(x, p\right)+x\frac{\mathrm{d}N\left(x, p\right)}{\mathrm{d}x}, \tag{33}$$

For evaluating the derivative term in the right hand side of (32), we use equa-

tions (6) and (26)–(31). The error function for this case may be formulated as

$$E(p)=\sum_{i=1}^{n}\left(\frac{\mathrm{d}w_t(x_i,p)}{\mathrm{d}x_i}-f\left(x_i,w_t(x_i,p)\right)\right)^2 \tag{34}$$

The weights from input to hidden are modified according to the following rule

$$w_{ji}^{r+1}=w_{ji}^{r}-\eta\left(\frac{\partial E}{\partial w_{ji}^{r}}\right) \tag{35}$$

where

$$\frac{\partial E}{\partial w_{ji}^{r}}=\frac{\partial}{\partial w_{ji}^{r}}\left(\sum_{i=1}^{n}\left(\frac{\mathrm{d}w_t(x_i,p)}{\mathrm{d}x_i}-f\left(x_i,w_t(x_i,p)\right)\right)^2\right) \tag{36}$$

Here, $\eta$ is the learning rate and $r$ is the iteration step. The weights from hidden to output layer may be updated in a similar formulation as done for input to hidden. Now going back to Equation (27), we recall that $z_j=\sum_{i=1}^{n}w_{ji}x_i+u_j$ and $N(x,p)=\sum_{j=1}^{m}v_j\sigma(z_j)$.This implies that;

$$\begin{aligned}\frac{\mathrm{d}N(x,p)}{\mathrm{d}x}&=\frac{\mathrm{d}}{\mathrm{d}x}\sum_{j}^{m}v_j\sigma\left(w_{ji}x_i+u_j\right)=\sum_{j}^{m}v_j\frac{\mathrm{d}}{\mathrm{d}x}\sigma\left(w_{ji}x_i+u_j\right)\\&=\sum_{j}^{m}v_j\frac{\mathrm{d}}{\mathrm{d}x}\sigma\left(z_j\right)=\sum_{j}^{m}v_j\frac{\mathrm{d}}{\mathrm{d}z_j}\sigma\left(z_j\right)\cdot\frac{\mathrm{d}z_j}{\mathrm{d}x}\end{aligned} \tag{37}$$

If the neural network model is a simple one as we saw in Section 3.3, then,

$$N(x,p)=\sigma(z)=\sigma\left(x_1w_1+x_2w_2+u\right)\ \Rightarrow\frac{\mathrm{d}N}{\mathrm{d}x_1}=\frac{\partial\sigma}{\partial z}\cdot\frac{\mathrm{d}z}{\mathrm{d}x_1};\ \frac{\mathrm{d}N}{\mathrm{d}x_2}=\frac{\partial\sigma}{\partial z}\cdot\frac{\mathrm{d}z}{\mathrm{d}x_2}$$

Now let us consider a first order differential equation:

$$\psi'(x)-\psi=0,\ x\in[0,1],\ \psi(0)=1 \tag{38}$$

with trial solution:

$$\psi_t(x,p)=A+xN(x,p) \tag{39}$$

where $x$ is the input to neural network model and $p$ represents the parameters—weights and biases.

$$w_t(0,p)=A+(0)N(0,p)=A=1\ \Rightarrow\psi_t(x,p)=1+xN(x,p) \tag{40}$$

To solve this problem using neural network (NN), we shall employ a NN architecture given in **Figure 3**. Now, $\psi_t(x_i,p)=1+x_iN(x_i,p)$, with $N(x_i,p)=\sum_{j=1}^{3}v_j\sigma(z_j)$, and

$$\sum_{j=1}^{3}v_j\sigma\left(z_j\right)=v_1\sigma\left(z_1\right)+v_2\sigma\left(z_2\right)+v_3\sigma\left(z_3\right)$$

where $z_1=x_1w_{11}+u_1$, $z_2=x_1w_{12}+u_2$ and $z_3=x_1w_{13}+u_3$.

If the neural network model is not able to predict correctly the solution of the differential equation with the given initial parameters—weights and biases, we

need to find the prediction error given by

$$E(p)=\sum_{i=1}^{n}\left(\frac{\mathrm{d}w_t(x_i,p)}{\mathrm{d}x_i}-f\left(x_i,w_t(x_i,p)\right)\right)^2. \tag{41}$$

If the prediction error does not satisfy an acceptable threshold, then the parameters need to be adjusted using the equation,

$$w_{ji}^{r+1}=w_{ji}^{r}-\eta\left(\frac{\partial E}{\partial w_{ji}^{r}}\right),\ \text{where}\ \ \frac{\partial E}{\partial w_{ji}^{r}}=\frac{\partial}{\partial w_{ji}^{r}}\sum_{i=1}^{n}\left(\frac{\mathrm{d}w_t(x_i,p)}{\mathrm{d}x_i}-f\left(x_i,w_t(x_i,p)\right)\right)^2 \tag{42}$$

Recall that: $\psi_t(x_i,p)=1+x_iN(x_i,p)$, and

$$\frac{\mathrm{d}w_t}{\mathrm{d}x_i}(x_i,p)=\frac{\mathrm{d}}{\mathrm{d}x_i}\left(1+x_iN(x_i,p)\right)=N(x_i,p)+x_i\frac{\mathrm{d}}{\mathrm{d}x_i}N(x_i,p). \tag{43}$$

$$\begin{aligned}\therefore\ \ \frac{\mathrm{d}}{\mathrm{d}x_1}N(x_1,p)&=\frac{\mathrm{d}}{\mathrm{d}x_1}\sum_{j=1}^{3}v_j\sigma(z_j)=v_j\frac{\mathrm{d}}{\mathrm{d}x_1}\sum_{j=1}^{3}\sigma(z_j)\\&=v_1\frac{\mathrm{d}\sigma}{\mathrm{d}z_1}\frac{\mathrm{d}z_1}{\mathrm{d}x_1}+v_2\frac{\mathrm{d}\sigma}{\mathrm{d}z_2}\frac{\mathrm{d}z_2}{\mathrm{d}x_1}+v_3\frac{\mathrm{d}\sigma}{\mathrm{d}z_3}\frac{\mathrm{d}z_3}{\mathrm{d}x_1}\end{aligned}$$

$$\frac{\mathrm{d}}{\mathrm{d}x_1}N(x_1,p)=v_1\sigma'(z_1)w_{11}+v_2\sigma'(z_2)w_{12}+v_3\sigma'(z_3)w_{13} \tag{44}$$

Putting Equations (42) and (43) into Equation (40) gives

$$E(p)=\sum_{i=1}^{n}\left(\left(N(x_i,p)+x_i\frac{\mathrm{d}}{\mathrm{d}x_i}N(x_i,p)\right)-\left(1+xN(x,p)\right)\right)^2$$

$$\therefore\ E(p)=\left(\sum_{j=1}^{3}v_j\sigma(z_j)+x_1\left(\frac{\mathrm{d}}{\mathrm{d}x_1}N(x_i,p)\right)-1-x_1\left(\sum_{j=1}^{3}v_j\sigma(z_j)\right)\right)^2 \tag{45}$$

$$\begin{aligned}\Rightarrow\ \ E(p)=&\left(v_1\sigma(z_1)+v_2\sigma(z_2)+v_3\sigma(z_3)\right.\\&+x_1\left(v_1\sigma'(z_1)w_{11}+v+v_2\sigma'(z_2)w_{12}+v_3\sigma'(z_3)w_{13}\right)\\&\left.-1-x_1\left(v_1\sigma(z_1)+v_2\sigma(z_2)+v_3\sigma(z_3)\right)\right)^2\end{aligned} \tag{46}$$

We noted earlier that when the neural network is not able to predict acceptable solution, that is, solution with minimum error, the weights and biases need to be adjusted. This involves complex derivatives with multivariable chain rule [11] [12]. From the ongoing, we need to compute the derivative of Equation (46) with respect to the weights and biases. That is:

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}^{r}}=&\frac{\partial}{\partial w_{ji}^{r}}\left(v_1\sigma(z_1)+v_2\sigma(z_2)+v_3\sigma(z_3)\right.\\&+x_1\left(v_1\sigma'(z_1)w_{11}+v+v_2\sigma'(z_2)w_{12}+v_3\sigma'(z_3)w_{13}\right)\\&\left.-1-x_1\left(v_1\sigma(z_1)+v_2\sigma(z_2)+v_3\sigma(z_3)\right)\right)^2\end{aligned} \tag{47}$$

Similarly, to update the weights from the hidden layer to output layer, we compute $\partial E/\partial v_j$, $j=1,2,3$. Finally, we update the biases by computing $\partial E/\partial u_j$, $j=1,2,3$. Equation (47) together with $\partial E/\partial v_j$ and $\partial E/\partial u_j$ are used

to update the weights and biases. The superscript $(r)$ denotes the $r^{\text{th}}$ iteration.

It is important to note that the foregoing is necessary in order to achieve the number of iterations required. Sometimes, it may be necessary to do up to 30 or more iterations for the solution to converge within an acceptable threshold. It is on the basis of this complex derivatives involved in solving ODE with neural network, especially of higher order, and the many iterations required for convergence that motivated our search for a more efficient and accurate way of solving the given problem, but avoiding the inherent computational complexity.

### 5.3. Results

We begin with a couple of examples on first and second order differential equations.

### 5.4. Example

Consider the initial value problem;

$$y' - \alpha y = 0;\ y(0) = 1,\ x \in [0,1],\ \alpha \equiv 1. \tag{48}$$

This equation has been solved by Mall and Chakraverty [13]. As discussed in the previous section, the trial solution is given by;

$$y_t(x) = A + xN(x, p)$$

Applying the initial condition gives $A = 1$, therefore $y_t(x) = 1 + xN(x, p)$.

To obtain the weights from input to hidden layer, it is natural to assume $y_a(x) = \mathrm{e}^x$. We approximate this function using regression in SPSS. We shall train the network for 10 equidistant points in [0, 1], and then employ excel spreadsheet to find values of $y_a(x) = \mathrm{e}^x$ at all the points. This leads us to the following data in **Table 3**.

We then perform a curve-fit polynomial regression built into IBM SPSS 23 [14], for the function $y_a(x) = \mathrm{e}^x$. The output is displayed below in **Table 4**.

**Table 4** displays the linear, quadratic and cubic regression of $y_a(x) = \mathrm{e}^x$. The quadratic and cubic curves show perfect *goodness of fit*, $R^2 = 1$. Using the cubic curve, we pick our weights from input layer to hidden layer as:

**Table 3.** Values of $\left(x, y_a(x) = \mathrm{e}^x\right)$ for problem (47).

| $X$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $Y$ | 1 | 1.1052 | 1.2214 | 1.3499 | 1.4918 | 1.6487 | 1.8221 | 2.014 | 2.226 | 2.4596 | 2.7183 |

**Table 4.** Model summary and parameter estimates.

| Equation | $R^2$ | Constant | $b_1$ | $b_2$ | $b_3$ |
|---|---|---|---|---|---|
| Linear | 0.981 | 0.883 | 1.698 | | |
| Quadratic | 1.000 | 1.010 | 0.856 | 0.842 | |
| Cubic | 1.000 | 1.000 | 1.016 | 0.423 | 0.279 |

The Independent variable is $X$

$w_{11} = 1.016$, $w_{12} = 0.423$, $w_{13} = 0.279$. Now to compute the weights from hidden layer to the output layer, we find a function $\vartheta(x)$ such that $v = \phi^{-1} f$; $v, f$ and $\phi$ are as defined in Section 3. In particular, $f(x) = (\vartheta(x_1), \vartheta(x_2), \vartheta(x_3))^{\mathrm{T}}$. We now form a linear function based on the default sign of the differential equation, *i.e.* $\vartheta(x) = ax - b$, where $a$ is the coefficient of the derivative of $y$ and $b$ is the coefficient of $y$ Thus;

$$\vartheta(x) = x + 1, \quad f(x) = (\vartheta(x_1), \vartheta(x_2), \vartheta(x_3))^{\mathrm{T}} = (1.1, 1.2, 1.3)^{\mathrm{T}}$$

The neural architecture for the neural network is shown in **Figure 2**, so we let $N = 3$. We take $x = (0.1, 0.2, 0.3)^{\mathrm{T}}$ and $f(x) = (1.1, 1.2, 1.3)^{\mathrm{T}}$. It then follows that

$$v = \phi^{-1} f, \ \Rightarrow \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \varphi_3(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \varphi_3(x_2) \\ \varphi_1(x_3) & \varphi_2(x_3) & \varphi_3(x_3) \end{bmatrix}^{-1} \begin{bmatrix} \vartheta_1 \\ \vartheta_2 \\ \vartheta_3 \end{bmatrix} \tag{49}$$

where

$$\varphi_i(x_j) = \exp\left( -\frac{\left(\left| x_i - x_j \right|\right)^2}{2\sigma^2} \right), \quad i = 1, 2, 3; \ \ j = 1, 2, 3. \tag{50}$$

Substituting the given values of the vectors $x$ and $f$, we obtain the weights from the hidden layer to the output layer,

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.94 & 0.78 \\ 0.94 & 1 & 0.94 \\ 0.78 & 0.94 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1.1 \\ 1.2 \\ 1.3 \end{bmatrix} = \begin{bmatrix} 5.17 \\ -9.375 \\ 6.08 \end{bmatrix}, \tag{51}$$

Therefore the weights from the hidden layer to the output layer are;
$v_1 = 5.17$, $v_2 = -9.375$, $v_3 = 6.08$.

The biases are fixed between −1 and 1. We now train the network with the available parameters using MathCAD 14 software [15] as follows:

$w_1 := 1.016 \quad w_2 := 0.423 \quad w_3 := 0.279 \quad x := 1$

$v_1 := 5.17 \quad v_2 := 6.08 \quad u_1 := 1 \quad u_2 := 0.2251 \quad u_3 := -0.1$

$z_1 := w_1 \cdot x + u_1 = 2.016 \quad z_2 := w_2 \cdot x + u_2 = 0.6481 \quad z_3 := w_3 \cdot x + u_3 = 0.179$

$\sigma(z_1) := \left[1 + \exp(z_1)\right]^{-1} = 0.882467, \ \sigma(z_2) := \left[1 + \exp(z_2)\right]^{-1} = 0.656582,$

$\sigma(z_3) := \left[1 + \exp(z_3)\right]^{-1} = 0.544631$

$N := v_1 \cdot \sigma(z_1) + v_2 \cdot \sigma(z_2) + v_3 \cdot \sigma(z_3) = 1.718251$

$y_p(x) := 1 + x \cdot N = 2.718251, \ \ y_d(x) := e^x = 2.718282$

$E := 0.5 \cdot \left(y_d(x) - y_p(x)\right)^2 = 4.707964 \times 10^{-10}$

Here $y_d$ and $y_p$ are respectively the desired output (exact solution) and the predicted output (trial solution). From the indicated error value, this is an acceptable accuracy. We compare our results with the neural network results obtained by Otadi and Mosleh [16] and find them to be in reasonable agreement.

This is depicted in Table 5, as well as the graphical profile in Figure 4 below.

The perfect accuracy is evident in the graphical profile depicted in Figure 4.

### 5.5. Remark

In what follows, we consider a non-homogeneous second order linear differential equation. It is important to recall that for any second order non-homogeneous differential equation of the form $y''(x)+a(x)y'+b(x)y=f(x)$, the non-homogeneous term $f(x)$ is termed the *forcing* function. In this section, we shall employ the forcing function to compute the weights from hidden layer to the output layer. This is made clear in the following example.

### 5.6. Example

Consider the initial value problem;

$$y''-4y=24\cos(2x);\ y(0)=3,\ y'(0)=4,\ x\in[0,1].$$

The trial solution is $y_t(x)=A+Bx+x^2N(x,p)$. Applying the initial condition gives $A=3, B=4$.

Therefore, $y_t(x)=3+4x+x^2N(x,p)$, $y_a(x)=e^{-2x}+e^{-3x}$.

We use excel spreadsheet to find values of $y_a(x)$ at all the $x$ points, as displayed in Table 6.

Using regression and SPSS model we find weights from input layer to hidden layer. From Table 7 and using the cubic curve fit with $R^2=1$, we pick our weights from input layer to hidden layer as:

$$w_{11}=0.488,\ w_{12}=1.697,\ w_{13}=3.338.$$

Now to compute the weights from hidden layer to the output layer, we use the function:

Table 5. Comparison of the results.

| Input data (X) | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Y Exact | 1 | 1.1052 | 1.2214 | 1.3499 | 1.4918 | 1.6487 | 1.8221 | 2.0138 | 2.226 | 2.46 | 2.7183 |
| Y Pred | 1 | 1.1052 | 1.2214 | 1.3499 | 1.4918 | 1.6488 | 1.8222 | 2.0137 | 2.226 | 2.46 | 2.7183 |

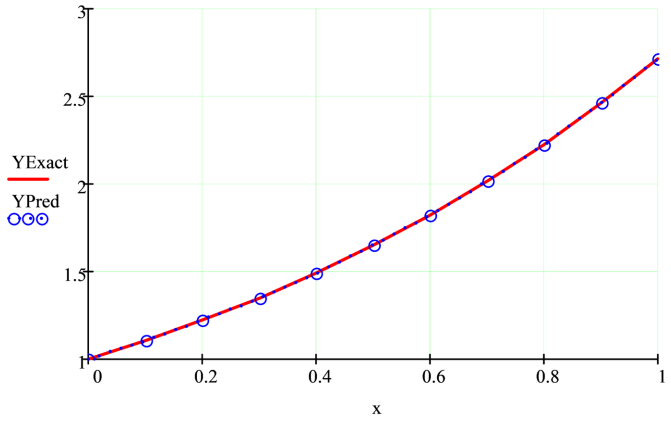


Figure 4. Plot of Y exact and Y predicted for Example 1.

$$\vartheta(x) = 24\cos(2x),$$

$$f(x) = \left(\vartheta(x_1), \vartheta(x_2), \vartheta(x_3)\right)^{\mathrm{T}} = \left(23.999854, 23.999415, 23.998684\right)^{\mathrm{T}}$$

With $x = (0.1, 0.2, 0.3)^{\mathrm{T}}$. Hence, the weights from the hidden layer to the output layer given by $v = \phi^{-1} f$ are;

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.94 & 0.78 \\ 0.94 & 1 & 0.94 \\ 0.78 & 0.94 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 23.999854 \\ 23.999415 \\ 23.998684 \end{bmatrix} \Rightarrow \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 112.489 \\ -187.474 \\ 112.483 \end{bmatrix}$$

The weights from the hidden layer to the output layer are;

$v_1 = 112.489$, $v_2 = -187.474$, $v_3 = 112.483$.

The biases are fixed between −1 and 1. We now train the network with the available parameters using our MathCAD 14 algorithm as follows:

$$w_1 := 0.488 \quad w_2 := 1.697 \quad w_3 := 3.338 \quad x := 1$$

$$v_1 := 112.489 \quad v_2 := -187.474 \quad v_3 := 112.483 \quad u_1 := 1 \quad u_2 := 1 \quad u_3 := -0.1691$$

$$z_1 := w_1 \cdot x + u_1 = 1.488 \quad z_2 := w_2 \cdot x + u_2 = 2.697 \quad z_3 := w_3 \cdot x + u_3 = 3.1689$$

$$\sigma(z_1) := \left[1 + \exp(-z_1)\right]^{-1} = 0.815778, \ \sigma(z_2) := \left[1 + \exp(-z_2)\right]^{-1} = 0.936849,$$

$$\sigma(z_3) := \left[1 + \exp(-z_3)\right]^{-1} = 0.959647 \quad .$$

$$N := v_1 \cdot \sigma(z_1) + v_2 \cdot \sigma(z_2) + v_3 \cdot \sigma(z_3) = 24.075112$$

$$y_p(x) := 3 + 4 \cdot x + x^2 \cdot N = 31.075112,$$

$$y_d(x) := 4 \cdot e^{2 \cdot x} + 2 \cdot e^{-2 \cdot x} - 3 \cdot \cos(2 \cdot x) = 31.075335$$

$$E := 0.5 \cdot \left(y_d(x) - y_p(x)\right)^2 = 2.502862 \times 10^{-8}$$

We compare the exact and approximate solution in Table 8. The accuracy is clearly depicted graphically in Figure 5.

**Table 6.** Values of $\left(x, y_a(x) = e^{2x} + e^{-2x}\right)$.

| $X$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $Y$ | 2 | 2.0401 | 2.1621 | 2.3709 | 2.67487 | 3.0862 | 3.6213 | 4.3018 | 5.1549 | 6.2149 | 7.524 |

**Table 7.** Model summary and parameter estimates.

| Equation | $R^2$ | Constant | $b_1$ | $b_2$ | $b_3$ |
|---|---|---|---|---|---|
| Linear | 0.887 | 1.630 | 5.283 | | |
| Quadratic | 0.995 | 1.931 | -3.698 | 6.703 | |
| Cubic | 1.000 | 1.985 | 0.488 | 1.697 | 3.338 |

The Independent variable is $X$

**Table 8.** Comparison of the results.

| **Input data (X)** | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Y Exact** | 3 | 3.6408 | 4.7630 | 6.3668 | 8.4520 | 11.0188 | 14.0670 | 17.5968 | 21.6081 | 26.1008 | 31.0751 |
| **Y Pred** | 3 | 3.5829 | 4.5448 | 5.9101 | 7.7107 | 9.9879 | 12.7958 | 16.2041 | 20.3035 | 25.2108 | 31.0753 |

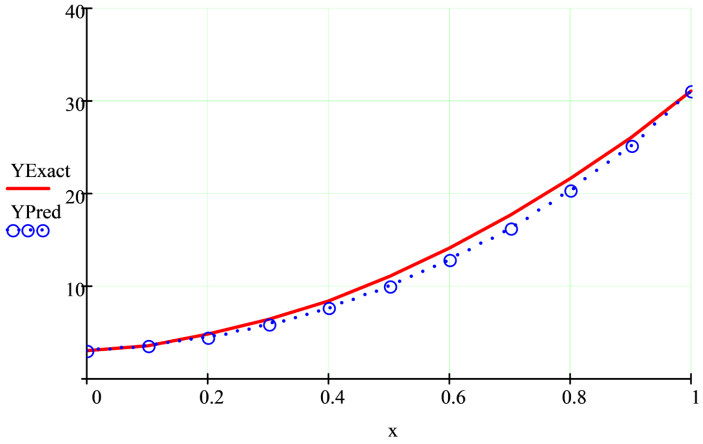


**Figure 5.** Plot of Y exact and Y predicted for Example 2.

## 6. Conclusion

In this paper, we have presented a novel approach for solving first and second order linear ordinary differential equations with constant coefficients. Specifically, we employ a feed-forward Multilayer Perceptron Neural Network (MLPNN), but avoid the standard back-propagation algorithm for updating the intrinsic weights. This greatly reduces the computational complexity of the given problem. Our results are validated by the near perfect approximations achieved in comparison with the exact solutions, as well as demonstrating the function approximation capabilities of ANN. This then proves the efficiency of our neural network procedure. We employed Excel spreadsheet, IBM SPSS 23, and MathCAD 14 algorithm to achieve this task.

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] McCulloch, W.S. and Pitts W. (1943) A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, **5**, 115-133. https://doi.org/10.1007/BF02478259

[2] Neumann, J.V. (1951) The General and Logical Theory of Automata. Wiley, New York.

[3] Graupe, D. (2007) Principles of Artificial Neural Networks. Vol. 6, 2nd Edition, World Scientific Publishing Co. Pte. Ltd., Singapore.

[4] Rumelhart, D.E. and McClelland, J.L. (1986) Parallel Distributed Processing, Explorations in the Microstructure of Cognition I and II. MIT Press, Cambridge. https://doi.org/10.7551/mitpress/5236.001.0001

[5] Werbos, P.J. (1974) Beyond Recognition, New Tools for Prediction and Analysis in the Behavioural Sciences. Ph.D. Thesis, Harvard University, Cambridge.

[6] Lagaris, I.E., Likas, A.C. and Fotiadis, D.I. (1997) Artificial Neural Network for Solving Ordinary and Partial Differential Equations. arXiv: physics/9705023v1.

[7] Cybenco, G. (1989) Approximation by Superposition of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, **2**, 303-314. https://doi.org/10.1007/BF02551274

[8] Hornic, K., Stinchcombe, M. and White, H. (1989) Multilayer Feedforward Networks Are Universal Approximators. *Neural Networks*, **2**, 359-366. https://doi.org/10.1016/0893-6080(89)90020-8

[9] Lee, H. and Kang, I.S. (1990) Neural Algorithms for Solving Differential Equations. *Journal of Computational Physics*, **91**, 110-131. https://doi.org/10.1016/0021-9991(90)90007-N

[10] Majidzadeh, K. (2011) Inverse Problem with Respect to Domain and Artificial Neural Network Algorithm for the Solution. *Mathematical Problems in Engineering*, **2011**, Article ID: 145608, 16 p. https://doi.org/10.1155/2011/145608

[11] Chen, R.T.Q., Rubanova, Y., Bettencourt, J. and Duvenaud, D. (2018) Neural Ordinary Differential Equations. arXiv: 1806.07366v1.

[12] Okereke, R.N. (2019) A New Perspective to the Solution of Ordinary Differential Equations using Artificial Neural Networks. Ph.D Dissertation, Mathematics Department, Michael Okpara University of Agriculture, Umudike.

[13] Mall, S. and Chakraverty, S. (2013) Comparison of Artificial Neural Network Architecture in Solving Ordinary Differential Equations. *Advances in Artificial Neural Systems*, **2013**, Article ID: 181895. https://doi.org/10.1155/2013/181895

[14] IBM (2015) IBM SPSS Statistics 23 http://www.ibm.com

[15] PTC (Parametric Technology Corporation) (2007) Mathcad Version 14. http://communications@ptc.com

[16] Otadi, M. and Mosleh, M. (2011) Numerical Solution of Quadratic Riccati Differential Equations by Neural Network. *Mathematical Sciences*, **5**, 249-257.