Scientific Research Publishing

Applied Mathematics, 2021, 12, 919-930
https://www.scirp.org/journal/am
ISSN Online: 2152-7393
ISSN Print: 2152-7385

Solving Riccati-Type Nonlinear Differential Equations with Novel Artificial Neural Networks

Roseline N. Okereke, Olaniyi S. Maliki

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

How to cite this paper: Okereke, R.N. and Maliki, O.S. (2021) Solving Riccati-Type Nonlinear Differential Equations with Novel Artificial Neural Networks. *Applied Mathematics*, **12**, 919-930.
https://doi.org/10.4236/am.2021.1210060

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

Abstract

In this study we investigate neural network solutions to nonlinear differential equations of Ricatti-type. We employ a feed-forward Multilayer Perceptron Neural Network (MLPNN), but avoid the standard back-propagation algorithm for updating the intrinsic weights. Our objective is to minimize an error, which is a function of the network parameters *i.e.*, the weights and biases. Once the weights of the neural network are obtained by our systematic procedure, we need not adjust all the parameters in the network, as postulated by many researchers before us, in order to achieve convergence. We only need to fine-tune our biases which are fixed to lie in a certain given range, and convergence to a solution with an acceptable minimum error is achieved. This greatly reduces the computational complexity of the given problem. We provide two important ODE examples, the first is a Ricatti type differential equation to which the procedure is applied, and this gave us perfect agreement with the exact solution. The second example however provided us with only an acceptable approximation to the exact solution. Our novel artificial neural networks procedure has demonstrated quite clearly the function approximation capabilities of ANN in the solution of nonlinear differential equations of Ricatti type.

Keywords

Ricatti ODE, MLPNN, GRBF, Network Training, MathCAD 14

1. Introduction

We present a new perspective for obtaining solutions of initial value problems of Ricatti-type [1], using Artificial Neural Networks (ANN). This is an extension of

the procedure developed by Okereke [2]. 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 differential equations 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.

1.1. Neural Network Structure

A neural network is an inter-connection of processing elements, units or nodes, whose functionality resemble that of the human neurons. 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. Training of the neural network can be done either by any optimization technique which in turn requires the computation of the gradient of the error with respect to the network parameters, by regression based model or by basis function approximation.

1.2. Neural Networks are Universal Approximators

Artificial neural network 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 has been find out 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.

1.3. Universal Approximation Theorem

The universal approximation theorem for MLP was proved by Cybenko [3] and Hornik *et al.* [4] 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 sigmo-

id function $\varphi(\cdot)$, then the universal approximation theorem states that the finite sums of the form

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

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

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

1.4. Learning in Neural Networks

A neural network has to be configured such that the application of a set of inputs produces the desired set of outputs. Various methods to set the strengths of the connection exist. One way is to set the weights explicitly, using priory knowledge. Another way is to train the neural network by feeding it, teaching patterns and letting it change its weights according to some learning rule. The term learning is widely used in the neural network field to describe this process; it might be formally described as: determining an optimized set of weights based on the statistics of the examples. The learning classification situations in neural networks may be classified into distinct sorts of learning: supervised learning, unsupervised learning, reinforcement learning and competitive learning [5].

1.5. 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 [6] 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 output of the network 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^{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^{th} order derivative of the sigmoid activation function.

2. General Formulation for Differential Equations

Let us consider the following general differential equations which represent both

ordinary and partial differential equations Majidzadeh [7]:

$$G\left(x,\psi\left(x\right),\nabla\psi\left(x\right),\nabla^2\psi\left(x\right),\cdots\right)=0,\ \forall x\in D, \tag{5}$$

subject to some initial or boundary conditions, where $x=\left(x_1,x_2,\cdots,x_n\right)\in\mathbb{R}^n$, $D\subset\mathbb{R}^n$ denotes the domain, and $\psi\left(x\right)$ is the unknown scalar-valued solution to be computed. Here, G is the function which defines the structure of the differential equation and ∇ is a differential operator. Let $\psi_t\left(x,p\right)$ denote the trail solution with parameters (weights, biases) p. Legaris *et al.* [8] gave the following as the general formulation for the solution of differential Equations (4) using ANN. Now, $\psi_t\left(x,p\right)$ may be written as the sum of two terms

$$\psi_t\left(x,p\right)=A\left(x\right)+F\left(x,N\left(x,p\right)\right) \tag{6}$$

where $A\left(x\right)$ satisfies initial or boundary condition and contains no adjustable parameters, whereas $N\left(x,p\right)$ is the output of feed forward neural network with the parameters p and input data x. The function $F\left(x,N\left(x,p\right)\right)$ 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 $N\left(x,p\right)$ is given by

$$N\left(x,p\right)=\sum_{j=1}^{m}v_j\sigma\left(z_j\right),\ \ z_j=\sum_{i=1}^{n}w_{ji}x_i+u_j\ . \tag{7}$$

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.

2.1. 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

$$\sigma\left(z_j\right)=\left(1+\mathrm{e}^{-z_j}\right)^{-1}. \tag{8}$$

2.2. Neural Network Model for Solving First Order Nonlinear ODE

Let us consider the first order ordinary differential equation below

$$\psi'\left(x\right)=f\left(x,\psi\right),\ x\in\left[a,b\right] \tag{9}$$

with initial condition $\psi\left(a\right)=A$. In this case we assume the function f is nonlinear in its argument. The ANN trial solution may be written as

$$\psi_t\left(x,p\right)=A+xN\left(x,p\right), \tag{10}$$

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. To solve this problem using neural network (NN), we shall employ a NN

architecture with three layers. One input layer with one neuron; one hidden layer with n neurons and one output layer with one output unit, as depicted in Figure 1 below.

Each neuron is connected to other neurons of the previous layer through adaptable synaptic weights w_{1j} and biases u_j. Now, $\psi_t(x_i, p) = A + x_i N(x_i, p)$ with

$$N(x, p) = \sum_{j=1}^{n} v_j \sigma(x w_j + u_j), \quad z_j = x w_j + u_j. \tag{11}$$

It is possible to have Multi-layered perceptrons with more than three layers, in which case we have more hidden layers [9] [10]. The most important application of multilayered perceptrons is their ability in function approximation. The Kolmogorov existence theorem guarantees that a three-layered perceptron with $n(2n+1)$ nodes can compute any continuous function of n variables [11] [12]. The accuracy of the approximation depends only on the number of neurons in the hidden layer and not on the number of the hidden layers [13]. For the purpose of numerical computation, as mentioned previously, our sigmoidal activation function $\sigma(\cdot)$ for the hidden units of our neural network is taken to be;

$$\sigma(z) = \left(1 + \mathrm{e}^{-z}\right)^{-1} \tag{12}$$

with the property that;

$$\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right). \tag{13}$$

The trial solution $\psi_t(x, p)$ satisfies the initial condition. We differentiate the trial solution $\psi_t(x, p)$ to get

$$\frac{\mathrm{d}\psi_t(x, p)}{\mathrm{d}x} = N(x, p) + x\frac{\mathrm{d}N(x, p)}{\mathrm{d}x}, \tag{14}$$

We observe that;

$$\frac{\mathrm{d}N(x, p)}{\mathrm{d}x} = \sum_{j=1}^{n} v_j \frac{\mathrm{d}}{\mathrm{d}x}\sigma(x w_j + u_j) = \sum_{j=1}^{n} v_j w_j \sigma'(z_j)$$

$$\Rightarrow \quad \frac{\mathrm{d}N(x, p)}{\mathrm{d}x} = \sum_{j=1}^{n} v_j w_j \sigma(z_j)\left(1 - \sigma(z_j)\right)$$

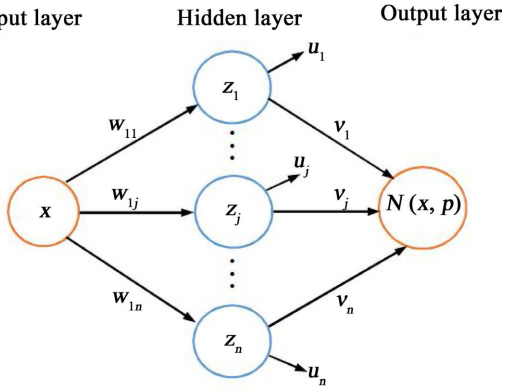

Figure 1. Schematic for $N(x, p)$.

For evaluating the derivative term in the right hand side of (32), we use equations (7) and (26)-(31).

The error function for this case is formulated as;

$$E(p)=\sum_{i=1}^{n}\left(\frac{\mathrm{d}\psi_t(x_i,p)}{\mathrm{d}x_i}-f\left(x_i,\psi_t(x_i,p)\right)\right)^2. \tag{15}$$

Minimization of the above error function is considered as a procedure for training the neural network, where the error corresponding to each input vector x is the value $f(x)$ 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 [14]. This process can be quite tedious computationally, and in this work we avoid this cumbersome process by introducing the novel procedure outlined in this paper.

3. Numerical Example

The Riccati equation is a nonlinear ordinary differential equation of first order of the form:

$$y'(x)=p(x)y+q(x)y^2+r(x) \tag{16}$$

where $p(x),q(x),r(x)$ are continuous functions of *x*. Neural network method can also solve this type of ODE. We show how our new approach can solve this type of ODE by redefining the neural network with respect to the form the ODE takes. Specifically, we consider the initial value problem:

$$y'(x)=2y(x)-y^2(x)+1,\ y(0)=0,\ x\in[0,1], \tag{17}$$

which was solved by Otadi and Mosleh (2011) [15]. The exact solution is $y(x)=\sqrt{2}\tanh\left(x\sqrt{2}\right)$.

The trial solution is given by $y_t(x)=A+x\aleph(x,p)$. Applying the initial conditions gives $A=0$. Therefore $y_t(x)=x\aleph(x,p)$. This solution obviously satisfies the given initial condition. We observe that in Equation (17), the term $y^2(x)$ is what makes the ODE nonlinear. Also this term cannot be separated from $2y(x)$. Therefore, we incorporate $2y(x)-y^2(x)$ into the neural network to take care of the nonlinearity seen in the given differential equation. Thus, the new neural network becomes,

$$\aleph(x,p)=\sum_{j}^{m}v_j\left[2\sigma(z_j)-\sigma^2(z_j)\right]=\sum_{j}^{m}v_j\sigma(z_j)\left[2-\sigma(z_j)\right] \tag{18}$$

The error to be minimized is

$$E=\frac{1}{2}\sum_{i=1}^{n}\left\{\frac{\mathrm{d}}{\mathrm{d}t}y_t(x_i,p)-\left[2y_t(x_i,p)-y_t^2(x_i,p)+1\right]\right\}^2 \tag{19}$$

where the set $\{x_i,i=1,\cdots,n\}$ are the discrete points in the interval $[0,1]$. We

proceed as follows.

To compute the weights $w_j, j=1,2,3$ from the input layer to the hidden layer (Figure 1), we construct a function $\vartheta(x)$ such that $w=\phi^{-1}f$, f and ϕ. In particular, for $\boldsymbol{x}=(x_1,x_2,x_3)$, $f(\boldsymbol{x})=(\vartheta(x_1),\vartheta(x_2),\vartheta(x_3))^{\mathrm{T}}$. Here $N=3$ and the solution $w=\phi^{-1}f$ is given by;

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \varphi_3(x_1) \\ \varphi_1(x_2) & \varphi_2(x_2) & \varphi_3(x_2) \\ \varphi_1(x_3) & \varphi_2(x_3) & \varphi_3(x_3) \end{bmatrix}^{-1} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{20}$$

Here;

$$\varphi_i(x)=\exp\left(-\frac{|x-x_i|^2}{2\sigma^2}\right),\ \sigma^2=\frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2,\ \bar{x}=\frac{1}{N}\sum_{i=1}^{N}x_i \tag{21}$$

The above is the so-called Gaussian Radial Basis function (GRBF) approximation model. To obtain the weights $v_j, j=1,2,3$ from hidden layer to the output layer, we construct another function $\theta(x)$ such that $v=\phi^{-1}f$, where, $f(\boldsymbol{x})=(\theta(x_1),\theta(x_2),\theta(x_3))^{\mathrm{T}}$, $\boldsymbol{x}=(x_1,x_2,x_3)$ and ϕ is given in Equation (20). We only need to replace the w_j's by the v_j's, $j=1,2,3$.

The exact form of $f(x)$ depends on the nature of a given differential equation. This will be made clear below. The nonlinear differential Equation (17) is rewritten as; $y'(x)-2y(x)+y^2(x)=1$.

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 (*i.e.* $a=1, b=-2$). Thus;

$$\vartheta(x)=x+2,\quad f(\boldsymbol{x})=(\vartheta(x_1),\vartheta(x_2),\vartheta(x_3))^{\mathrm{T}}=(2.1,2.2,2.3)^{\mathrm{T}},$$
$$\text{for}\quad \boldsymbol{x}=(0.1,0.2,0.3)^{\mathrm{T}}.$$

This we apply to get the weights from input layer to the hidden layer. Thus $f=(2.1,2.2,2.3)^{\mathrm{T}}$ and $w=\phi^{-1}f$

$$\Rightarrow \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.94 & 0.78 \\ 0.94 & 1 & 0.94 \\ 0.78 & 0.94 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 2.1 \\ 2.2 \\ 2.3 \end{bmatrix} \tag{22}$$

Hence, the weights from the input layer to the hidden layer are

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 41.335 & -73.437 & 36.79 \\ -73.437 & 139.062 & -73.437 \\ 36.79 & -73.437 & 41.335 \end{bmatrix} \begin{bmatrix} 2.1 \\ 2.2 \\ 2.3 \end{bmatrix},\ \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 9.858 \\ -17.187 \\ 10.767 \end{bmatrix} \tag{23}$$

The weights from input layer to the hidden layer are:

$w_1=9.858, w_2=-17.187, w_3=10.767$.

In order to get the weights from the hidden layer to the output layer, we now apply the forcing function which in this case is a constant function. That is, $\theta(x)=1$, which is a constant function.

$$\Rightarrow\quad \hat{f}=(\theta(x_1),\theta(x_2),\theta(x_3))^{\mathrm{T}}=(1,1,1)^{\mathrm{T}} \tag{24}$$

$\theta(x)$ being the nonhomogeneous term. With $v = \phi^{-1}\hat{f}$ the weights from the hidden layer to the output layer are given by

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0.94 & 0.78 \\ 0.94 & 1 & 0.94 \\ 0.78 & 0.94 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 41.335 & -73.437 & 36.79 \\ -73.437 & 139.067 & -73.437 \\ 36.79 & -73.437 & 41.335 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 4.687 \\ -7.812 \\ 4.687 \end{bmatrix} \tag{25}$$

Thus the weights from the hidden layer to the output layer are:

$v_1 = 4.687, v_2 = -7.812, v_3 = 4.687$.

The biases are fixed between −20 and 20. We now train the network with the available parameters using our MathCAD 14 [16] algorithm (computer output) as follows:

$w_1 := 9.858 \quad w_2 := -17.187 \quad w_3 := 10.767 \quad x := 1$

$v_1 := 4.687 \quad v_2 := -7.812 \quad v_3 := 4.687 \quad u_1 := -20 \quad u_2 := 10 \quad u_3 := -12.534$

$z_1 := w_1 \cdot x + u_1 = -10.142 \quad z_2 := w_2 \cdot x + u_2 = -7.187 \quad z_3 := w_3 \cdot x + u_3 = -1.767$

$\sigma(z_1) := \left[1 + \exp(z_1)\right]^{-1} = 3.9388 \times 10^{-5}, \ \sigma(z_2) := \left[1 + \exp(z_2)\right]^{-1} = 7.5578 \times 10^{-4},$

$\sigma(z_3) := \left[1 + \exp(z_3)\right]^{-1} = 0.1459$

$\aleph := v_1 \cdot \sigma(z_1) \cdot (2 - \sigma(z_1)) + v_2 \cdot \sigma(z_2) \cdot (2 - \sigma(z_2)) + v_3 \cdot \sigma(z_3) \cdot (2 - \sigma(z_3)) = 1.256457$

$y_p(x) := x \cdot \aleph = 1.256457, \qquad y_d(x) := \sqrt{2} \cdot \tanh\left(x \cdot \sqrt{2}\right) = 1.256367$

$E := 0.5 \cdot \left(y_d(x) - y_p(x)\right)^2 = 4.05 \times 10^{-4}$

The plots of the exact and predicted values in Table 1 are depicted in Figure 2 below.

Example

We consider the initial value problem:

$$x^2 y' + x^2 y^2 = 2, \ y\left(\frac{1}{2}\right) = 0, \ x \in (0,1] \tag{26}$$

The exact solution is easily computed as: $y(x) = \left(8x^3 - 1\right)\left(x + 4x^4\right)^{-1}$.

Our trial solution for the given problem is $y_t(x) = A + x\aleph(x, p)$. Applying the initial conditions gives

$$A = -\frac{1}{2}\aleph\left(\frac{1}{2}, p\right). \text{ Therefore, } \ y_t(x) = -\frac{1}{2}\aleph\left(\tfrac{1}{2}, p\right) + x\aleph(x, p) \tag{27}$$

In Equation (26), the nonlinear term $y^2(x)$ is alone in the ode (*i.e.* dividing

Table 1. 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	0	0.19868	0.38967	0.56642	0.72434	0.86106	0.97623	1.07104	1.14761	1.20852	1.25637
Y Pred	0	0.19867	0.38967	0.56642	0.72433	0.86106	0.97622	1.07103	1.14764	1.20849	1.25639

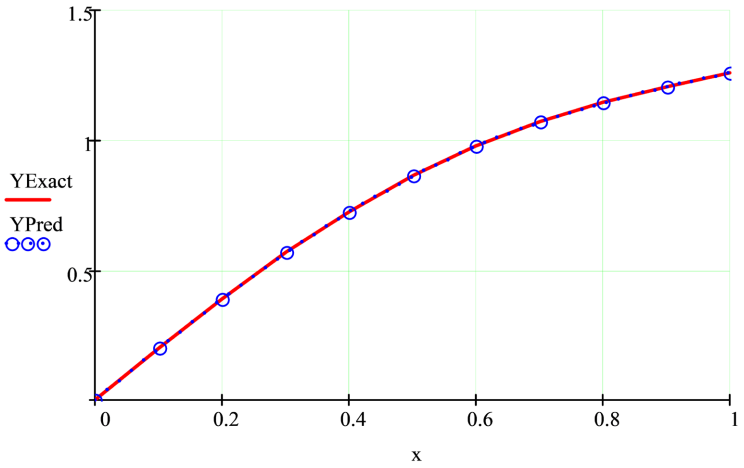

Figure 2. Plot of Y Exact and Y Predicted.

out rightly by x^2). Therefore, our neural network for this problem takes the form:

$$\aleph(x,p)=\sum_{j}^{3}v_j\sigma^2\left(z_j\right)=\sum_{j}^{3}v_j\sigma\left(z_j\right)\left[\sigma\left(z_j\right)\right] \tag{28}$$

We form algebraic equation of degree one with the default sign of the ode. Thus $\vartheta(x)=ax+b$, ($a=x^2, b=0$). Hence

$\vartheta(x)=x^3 \Rightarrow f(\boldsymbol{x})=(0.001,0.008,0.027)^{\mathrm{T}}$, for $\boldsymbol{x}=(0.1,0.2,0.3)^{\mathrm{T}}$

This we apply to get the weights from input layer to the hidden layer. We employ the GRBF here for the weights $w=\phi^{-1}f$. Hence;

$$\begin{bmatrix}w_1\\w_2\\w_3\end{bmatrix}=\begin{bmatrix}1 & 0.94 & 0.78\\0.94 & 1 & 0.94\\0.78 & 0.94 & 1\end{bmatrix}^{-1}\begin{bmatrix}0.001\\0.008\\0.027\end{bmatrix}\Rightarrow\begin{bmatrix}w_1\\w_2\\w_3\end{bmatrix}=\begin{bmatrix}0.447\\-0.944\\0.565\end{bmatrix} \tag{29}$$

The weights from input layer to the hidden layer are:

$w_1=0.447,\ w_2=-0.944,\ w_3=0.565$.

We now use the forcing function, a constant function in this case, to get the weights from the hidden layer to the output layer. That is,

$\theta(x)=2\Rightarrow\hat{f}(\boldsymbol{x})=(2,2,2)^{\mathrm{T}}$ for $\boldsymbol{x}=(0.1,0.2,0.3)^{\mathrm{T}}$. Hence, the weights $v=\phi^{-1}\hat{f}$ from the hidden layer to the output layer are;

$$\begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}=\begin{bmatrix}1 & 0.94 & 0.78\\0.94 & 1 & 0.94\\0.78 & 0.94 & 1\end{bmatrix}^{-1}\begin{bmatrix}2\\2\\2\end{bmatrix}\Rightarrow\begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}=\begin{bmatrix}9.375\\-15.625\\9.375\end{bmatrix} \tag{30}$$

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

$v_1=9.375,\ v_2=-15.625,\ v_3=9.375$.

The biases are fixed between −10 and 10. We now train the network with the available parameters using our MathCAD 14 algorithm as follows:

$w_1:=1.234 \quad w_2:=-2.725 \quad w_3:=1.716 \quad x:=1$

$v_1:=9.375 \quad v_2:=-15.625 \quad v_3:=9.375 \quad u_1:=-7 \quad u_2:=-4 \quad u_3:=-7$

$z_1:=w_1\cdot x+u_1=-5.766 \quad z_2:=w_2\cdot x+u_2=-6.725 \quad z_3:=w_3\cdot x+u_3=-5.284$

$$\sigma(z_1) := \left[1+\exp(z_1)\right]^{-1} = 0.998, \quad \sigma(z_2) := \left[1+\exp(z_2)\right]^{-1} = 0.995,$$

$$\sigma(z_3) := \left[1+\exp(z_3)\right]^{-1} = 0.998$$

$$\aleph(0.5) := v_1 \cdot \sigma(0.5 \cdot w_1 + u_1)^2 + v_2 \cdot \sigma(0.5 \cdot w_2 + u_2)^2 + v_3 \cdot \sigma(0.5 \cdot w_3 + u_3)^2 = 3.199$$

$$\aleph := v_1 \cdot \sigma(z_1)^2 + v_2 \cdot \sigma(z_2)^2 + v_3 \cdot \sigma(z_3)^2 = 3.01$$

$$y_p(x) := -0.5 \cdot \aleph(0.5) + x \cdot \aleph = 1.41, \quad y_d(x) := \left(8 \cdot x^3 - 1\right)\left(x + 4 \cdot x^4\right)^{-1} = 1.4,$$

$$E := 0.5 \cdot \left(y_d(x) - y_p(x)\right)^2 = 5 \times 10^{-5}$$

Table 2. Comparison of the results.

Input data (X)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1	1.1
Y Exact	−9.881	−4.535	−2.359	−0.971	0	0.651	1.050	1.269	1.371	1.4	1.3869
Y Pred	−1.245	−0.953	−0.66	−0.368	−0.075	0.218	0.51	0.803	1.095	1.388	1.68

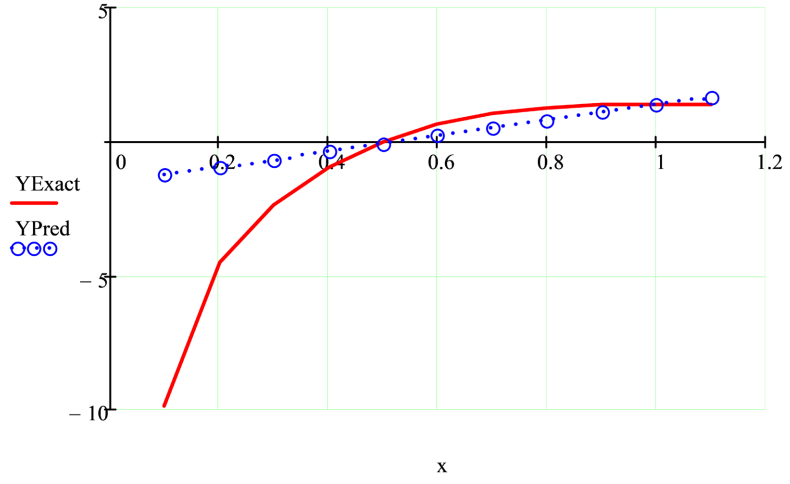

Figure 3. Plot of Y Exact and Y Pred.

The plots of the exact and predicted values in Table 2 are depicted in Figure 3.

4. Conclusion

A novel Neural Network approach was developed recently by Okereke, for solving first and second order linear ordinary differential equations. In this article, the procedure is now extended in this article to investigate neural network solutions to nonlinear differential equations of Ricatti-type. 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. For desired accuracy our objective is to minimize an error, which is a function of the network parameters *i.e.*, the weights and biases. Once the weights of the neural network are obtained by our systematic procedure, we need not adjust all the parameters in the network, as postulated by many researchers before us, in order

to achieve convergence. We only need to fine-tune our biases which are fixed to lie in a certain given interval, and convergence to a solution with an acceptable minimum error is achieved. The first example ODE of Ricatti type to which the procedure is applied gave us perfect agreement with the exact solution. The second example however provided us with only an acceptable approximation to the exact solution. This has demonstrated quite clearly the function approximation capabilities of ANN in the solution of nonlinear differential equations of Ricatti type. The above method still requires some refinement so that it can be generalized to solve any type of nonlinear differential equation including partial differential equations.

Conflicts of Interest

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

References

[1] Polyanin, A.D. and Zaitsev, V.F. (2003) Handbook of Exact Solutions for Ordinary Differential Equations. 2nd Edition, Chapman & Hall/CRC, Boca Raton.

[2] 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.

[3] 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

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

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

[6] 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

[7] 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

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

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

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

[11] Gurney, K. (1997) An Intorduction to Neural Networks. UCL Press, London.

[12] Samath, J.A., Kumar, P.S. and Begum, A. (2010) Solution of Linear Electrical Circuit Problem Using Neural Networks. *International Journal of Computer Applications*, **2**, 6-13. https://doi.org/10.5120/618-869

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

[14] Manoj, K. and Yadav, N. (2011) Multilayer Perceptrons and Radial Basis Function Neural Network Methods for the Solution of Differential Equations, A Survey. *Computers and Mathematics with Applications*, **62**, 3796-3811. https://doi.org/10.1016/j.camwa.2011.09.028

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

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