

Neural solver for sixth-order ordinary differential equations

Janavi Bhalala^{1,2} and B. Veena S. N. Rao ^{*3}

¹Department of Computer Science, The University of Texas at Dallas,
Richardson, TX-75080, USA

²Department of Computer Science, Texas A&M University-Corpus
Christi, Corpus Christi, TX-78412, USA

³Department of Mathematics & Statistics, Texas A&M University -
Corpus Christi Corpus Christi, TX 78412, USA

Abstract

A method for approximating sixth-order ordinary differential equations is proposed, which utilizes a deep learning feedforward artificial neural network, referred to as a neural solver. The efficacy of this unsupervised machine learning method is demonstrated through the solution of two distinct boundary value problems (BVPs), with the method being extended to include the solution of a sixth-order ordinary differential equation (ODE). The proposed mean squared loss function is comprised of two terms: the differential equation is satisfied by the first term, while the initial or boundary conditions are satisfied by the second. The total loss function is minimized using a quasi-Newton optimization method to obtain the desired network output. The approximation capability of the proposed method is verified for sixth-order ODEs. Point-wise comparisons of the approximations show strong agreement with available exact solutions. The proposed algorithm minimizes the overall learnable network hyperparameters in a given BVP. Simple minimization of the total loss function yields highly accurate results even with a low number of epochs. Therefore, the proposed framework offers an attractive setting for the computational mathematics community.

Keywords— Machine learning, Artificial neural networks, sixth-order ordinary differential equations, Optimizer, Activation Function

1 Introduction

The study of higher-order differential equations is a critical pursuit in numerous scientific and engineering fields, as these equations serve as the mathematical language for describing complex physical phenomena [1–5]. While analytical solutions are invaluable for their theoretical elegance, they are often nonexistent or exceedingly difficult to obtain for the nonlinear and

*Faculty Advisor

high-dimensional problems that frequently arise in real-world applications. This intractability necessitates a reliance on numerical methods. The development of robust and accurate numerical solvers for higher-order differential equations is therefore a fundamental and pressing challenge. These methods provide a powerful computational framework for approximating solutions to problems that lack closed-form expressions, thereby enabling detailed quantitative analysis and prediction. The ability to simulate and predict the behavior of systems, ranging from the intricate quantum mechanical interactions to the complex dynamics of fluid flow in biological and mechanical systems, hinges on our capacity to find high-fidelity numerical solutions. The pursuit of novel and more efficient numerical techniques is not merely an academic exercise; it is a vital endeavor that directly expands the boundaries of what is computationally possible, leading to significant advancements in scientific understanding, engineering design, and technological innovation. In this context, the search for novel and more efficient numerical techniques, such as those leveraging machine learning, represents a key frontier in computational science.

A significant and enduring challenge in computational science is the development of robust, efficient, and unified numerical methods for approximating solutions to complex differential equations. In recent years, Artificial Neural Networks (ANNs) have emerged as a powerful tool for this purpose, demonstrating a remarkable capability to serve as universal function approximators. This inherent strength is grounded in the Universal Approximation Theorem, first established in the seminal works of Cybenko and Hornik [6, 7], which proves that a feedforward network with a single hidden layer can approximate any continuous function on a compact subset of \mathbb{R}^n to any desired accuracy. The idea of leveraging ANNs as global approximators for differential equations was pioneered in the works of Lee and Kang [8], Meade and Fernandez [9], Logovski [10], and Lagaris [11]. These foundational studies established a framework where the output of a neural network is used to construct a discrete approximation of a differential equation's solution.

The use of ANNs as numerical solvers offers several key advantages over traditional methods. The resulting network approximation is inherently smooth and continuously differentiable, a property that is highly desirable in many physical applications. Furthermore, the ANN framework operates on a set of discrete, unstructured points, which completely bypasses the complex and often time-consuming process of meshing required by conventional techniques such as the finite element method. The network output is also less susceptible to common numerical errors, including rounding, domain discretization, and approximation errors. The inherent versatility of ANNs allows the framework to be easily tuned and adapted to approximate solutions for ordinary, partial, and integral equations [12–16], and even for approximating Green's functions for nonlinear elliptic operators [17]. These advantages are often accompanied by a reduced number of hyperparameters and the method's inherent suitability for parallel architectures.

Driven by the development of highly efficient deep learning libraries like TensorFlow and Keras [18], a considerable body of recent literature has been published on the development of new deep learning frameworks for solving differential equations. In the present work, we contribute to this field by examining the use of a deep learning feedforward neural network for the solution of nonlinear ordinary differential equations. The framework is implemented in Python, leveraging the computational power of TensorFlow and Keras. For the purpose of illustration, two distinct boundary value problems are considered: a sixth-order linear and a nonlinear ODE. In both problems, the core methodology involves the minimization of a total loss function, which yields a highly accurate, collocated approximation of the exact solution. The network solutions are then meticulously compared with the exact solutions.

This work builds upon the foundational research in our previous studies [19–22], with the specific aim of addressing theoretical gaps and enhancing the computational efficiency of neural network solvers. In this investigation, we introduce and apply a highly effective mean-square metric to evaluate the performance of the proposed deep learning feedforward neural network approximation method. A key feature of our approach is the formulation of a "hard method,"

where the total loss function is composed of two distinct and critical components: a differential cost term that enforces the governing differential equation, and a boundary loss term that explicitly satisfies the initial and boundary conditions. This dual-term loss function ensures that the network's output is not only a valid solution but also adheres precisely to the problem's constraints. A significant advantage of this computational implementation is that it does not require a closed-form analytical solution for the training process. Instead, the network is trained using a linear interpolation of data values to construct the tangent line and curvature, allowing the network's predictions to self-adjust to the differential equation. The overall training is further refined through backpropagation, where the network's hyperparameters are precisely fine-tuned to optimize the solution. This process yields a robust and highly accurate approximation, demonstrating the remarkable capacity of neural network frameworks to solve complex problems without the need for traditional, often limiting, numerical techniques.

The remainder of this paper is structured to systematically present our methodology and findings. Section 2 provides a comprehensive mathematical foundation, detailing the core method, the role of various activation functions, and a step-by-step implementation algorithm. In Section ??, we demonstrate the efficacy and robustness of the proposed method by applying it to two distinct numerical examples: a linear sixth-order ordinary differential equation (ODE) and a challenging nonlinear counterpart. Finally, Section 4 summarizes our conclusions and discusses promising avenues for future research.

2 Neural network architecture

A **Feedforward Neural Network (FFNN)** is employed for this study, taking an input vector $\mathbf{x} \in \mathbb{R}^n$ and producing m outputs. The network architecture consists of an input layer (l_1), two hidden layers (l_2 and l_3), and an output layer (l_4), as depicted in Figure 3. The input layer, l_1 , takes the input vector $\mathbf{x} = \{x_1, x_2, \dots, x_n\}$ and connects it to the H nodes of the first hidden layer, l_2 , via a weight matrix with individual weights denoted as w_{ij} . The connections between the hidden layers, l_2 and l_3 , are defined by weights \tilde{w}_{ij} , while the final hidden layer (l_3) is connected to the output layer (l_4) by weights v_{ij} . Each node within the network, including the hidden and output layers, receives an input that is the weighted sum of outputs from the previous layer, along with an additional term known as the **bias**. The output of the hidden layers is then processed by an **activation function**, and the final output of the network, denoted as $N(\mathbf{x}, \mathbf{v})$, provides the approximate solution. This multi-layered structure allows the network to learn complex, non-linear relationships between the input and output data.

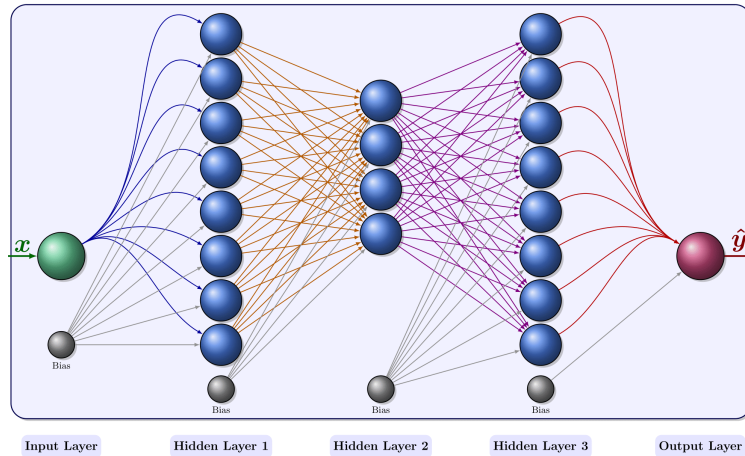


Figure 1: Feedforward neural network architecture with a single hidden layer.

The fundamental components that define the strength of connections within a neural network are its **weights** and **biases**, collectively referred to as the network parameters \mathbf{P} . The weights, in particular, determine the influence of one neuron's output on the next. A critical step in the training process is the initialization of these parameters. Initializing all weights to the same value can lead to a failure in learning, as the network's neurons would produce identical outputs and gradients, preventing the model from distinguishing different patterns. To circumvent this, the weights are initialized with random values drawn from specific distributions [23].

These weights are represented in matrix form as:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1j} \\ w_{21} & w_{22} & \cdots & w_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i1} & w_{i2} & \cdots & w_{ij} \end{bmatrix}. \quad (1)$$

The choice of an appropriate **initializer algorithm** is crucial, as the network's performance is highly sensitive to the initial values of its weights. Two widely used initializers are *Glorot uniform* and *Glorot normal* [24]. The former draws values from a uniform distribution, while the latter uses a normal distribution. For *Glorot normal*, the weights are initialized using a mean $\tilde{m} = 0$ and a standard deviation $\tilde{s} = \sqrt{\frac{2}{I_{in} + O_{out}}}$, where I_{in} and O_{out} are the number of input and output nodes, respectively. For *Glorot uniform*, the weights are initialized from a uniform distribution with a range of $[-\sqrt{\frac{6}{I_{in} + O_{out}}}, \sqrt{\frac{6}{I_{in} + O_{out}}}]$.

In addition to weights, the network also incorporates a special type of constant input known as the **bias** \mathbf{b} . Each neuron has its own bias, which is not influenced by the outputs of other neurons. These weights and biases constitute the complete set of network parameters \mathbf{P} that are iteratively adjusted during training to minimize the loss function and achieve the desired output.

2.1 The role of activation functions

The **activation function**, also known as a **squashing function**, is a fundamental component applied to each neuron in the hidden and output layers of the neural network. Its primary purpose is to introduce non-linearity into the network, enabling it to learn complex mappings and solve problems that cannot be addressed by a simple linear model. Mathematically, an activation function σ is defined as a non-decreasing, real-valued function that maps from \mathbb{R} to a bounded range, typically $[0, 1]$ or $[-1, 1]$ [25]. The selection of an activation function is a critical design choice, as it significantly impacts the network's learning capacity and performance.

Among the various activation functions, this work employs several common types: *Tanh*, *Sigmoid*, and *Softmax*. The **Sigmoid activation function**, $\sigma_s(\mathbf{x}) : \mathbb{R} \rightarrow [0, 1]$, is a bounded and differentiable function characterized by its distinctive "S"-shaped curve. It is defined as:

$$\sigma_s(\mathbf{x}) = \frac{1}{(1 + e^{-\mathbf{x}})} \quad (2)$$

Its derivative, which is essential for backpropagation, is given by:

$$\sigma'_s(\mathbf{x}) = \sigma_s(\mathbf{x})(1 - \sigma_s(\mathbf{x})) \quad (3)$$

The **Softmax activation function** is particularly useful for multi-class classification problems, as it takes an n -dimensional input vector and produces an n -dimensional output vector where each value lies between 0 and 1 and the sum of all values is equal to 1. This output can be interpreted as a probability distribution over the possible classes. The function and its

derivative are defined as:

$$\sigma_t(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad \forall i, j \in \{1, \dots, N\} \quad (4)$$

$$\sigma'_t(x_i) = \begin{cases} \sigma_t(x_i)(1 - \sigma_t(x_i)) & i = j, \\ -\sigma_t(x_i)\sigma_t(x_j) & i \neq j. \end{cases} \quad (5)$$

The mathematical operation within a hidden layer involves a linear combination of the inputs and their corresponding weights, which is then passed through an activation function. The output of the i -th hidden node is given by:

$$y_i = \sigma(g_i) = \sigma \left(\sum_{j=1}^n x_j w_{ij} \right). \quad (6)$$

This process is repeated for each hidden neuron, and the final output of the FFNN, denoted by N_j , is a linear combination of the activated outputs from the last hidden layer:

$$N_j = \sum_{i=1}^H v_{ij} y_i. \quad (7)$$

The choice of activation function is contingent on the specific problem. The final network output is a function of the input vector and the network parameters, $N(\mathbf{x}, \mathbf{P})$, and is subsequently used to define the loss function, which serves as the metric for evaluating the solution's accuracy.

2.2 The optimization problem and training process

The core of training a neural network lies in solving a **minimization problem**, where the objective is to find a set of network parameters \mathbf{P} that minimize the loss function $L(\mathbf{x}, \mathbf{P})$. This process is performed by **optimizers**, which are algorithms designed to iteratively update the parameters in the direction of steepest descent.

Definition 1 (Minimization Problem). *Let $\tilde{L} : \mathbb{R}^n \rightarrow \mathbb{R}$ be a continuously differentiable function and $\mathbf{x} \in \mathbb{R}^n$ be an n -dimensional vector subjected to certain constraints c . The minimization problem for \tilde{L} is then defined as follows:*

$$\min_{\mathbf{x} \in \mathbb{R}^n} \tilde{L}(\mathbf{x}) \quad \text{subject to} \quad \begin{cases} c_i(x_1, \dots, x_n) = r_i & i = 1, \dots, m \\ c_j(x_1, \dots, x_n) \geq s_j & j = 1, \dots, z. \end{cases} \quad (8)$$

Definition 2 (Minimum). *A vector of points \mathbf{x}^* is called a minimum of an objective function \tilde{L} if,*

$$\tilde{L}(\mathbf{x}^*) \leq \tilde{L}(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n. \quad (9)$$

The optimization process begins with an initial guess for the parameters and then generates a sequence of corrections until a minimum is reached. The primary role of an optimizer in a neural network is to find the optimal parameters that significantly reduce the loss function during training. A common approach is a **gradient descent-based optimization method**, which iteratively updates the parameters in the negative gradient direction of the loss function, ensuring that the parameters converge toward a minimum. The gradients are efficiently computed using the **backpropagation algorithm** [26], which propagates the error from the output layer back through the network, allowing for the precise adjustment of each weight and bias.

The overall workflow of the neural network model involves two distinct phases: **training** and **testing**. Initially, a training dataset is fed through the network's input layer to the hidden layers. The activation function is applied to each hidden node to produce the activated weighted sum, which is then used to define the loss function. This function serves as the primary metric for evaluating the neural network's solution quality. The training phase concludes once the optimizer has converged on a set of parameters that minimize the loss function. In the subsequent testing phase, the trained network model is evaluated on the same training data to generate the final numerical solutions, which are then used for post-processing and comparison with other results.

Here, we detail the mechanism of our neural network approach to approximate the solution to a sixth-order ODE. We consider the general form of a differential operator \mathcal{L} defined on the domain $\Omega = (a, b)$:

$$\mathcal{L}[x, y(x), y'(x), y''(x), \dots, y^{vi}(x)] = f(x) \quad \text{for } x \in \Omega : (a, b), \quad (10)$$

subject to given boundary conditions at $x = a$ and $x = b$. Our approach approximates the exact solution $y(x)$ with a neural network output $\hat{y}(\mathbf{x}, \mathbf{P})$, where \mathbf{x} is the input vector and \mathbf{P} represents the adjustable network parameters (weights and biases).

The primary objective of the training phase is to find the optimal parameters \mathbf{P} that minimize a carefully constructed **loss function** $L(\mathbf{x}, \mathbf{P})$. This loss function is a composite metric that evaluates how well the network's approximation satisfies both the differential equation and the boundary conditions. It consists of two main terms: a differential equation loss (L_d) and a boundary condition loss (L_{bc}).

The differential equation loss is formulated as a mean squared error, measuring the discrepancy between the network's output and the differential equation's right-hand side. This is calculated over a set of N_{int} interior data points:

$$L_d = \frac{1}{N_{int}} \sum_{i=1}^{N_{int}} (\mathcal{L}[x_i, \hat{y}(x_i), \hat{y}'(x_i), \hat{y}''(x_i), \dots, \hat{y}^{vi}(x_i)] - f(x_i))^2. \quad (11)$$

Simultaneously, the boundary condition loss ensures that the network's solution adheres to the prescribed values at the boundaries of the domain. Our approach implements this as a **"hard method,"** where the boundary constraints are directly embedded into the loss function. The boundary loss is given by:

$$L_{bc} = (\hat{y}(a) - y(a))^2 + (\hat{y}''(a) - y''(a))^2 + (\hat{y}^{iv}(a) - y^{iv}(a))^2 \\ + (\hat{y}(b) - y(b))^2 + (\hat{y}''(b) - y''(b))^2 + (\hat{y}^{iv}(b) - y^{iv}(b))^2. \quad (12)$$

The total loss function, which the network seeks to minimize, combines these two components into a single mean squared metric:

$$L = L_d^2 + L_{bc}^2. \quad (13)$$

This unified loss function allows the network to find an approximate solution that is not only valid across the domain but also strictly satisfies the boundary conditions, ensuring the physical consistency of the solution.

The overall process of the neural approximation consists of a training and testing phase that can be viewed as an algorithm :

Algorithm 1 Neural Network Solver for Boundary Value Problems

- 1: **Problem Formulation:** Define the sixth-order BVP in the form of a differential operator \mathcal{L} as shown in Equation (10), along with its corresponding boundary conditions.
 - 2: **Network Architecture Selection:** Intuitively choose a suitable feedforward neural network architecture, including the number of hidden layers, nodes, and activation functions, based on the complexity of the problem.
 - 3: **Parameter Initialization:**
 - a. Initialize all network parameters $\mathbf{P} = \{\mathbf{W}, \mathbf{b}\}$ (weights and biases) using a suitable random initializer (e.g., Glorot uniform or Glorot normal).
 - b. Initialize a loss variable L to a large value.
 - 4: **Training Loop:** Iterate for a predefined number of epochs or until $L \leq \epsilon$.
 - a. **Forward Pass:** Propagate the input data points through the network to compute the network's output $\hat{y}(\mathbf{x}, \mathbf{P})$.
 - b. **Loss Computation:** Calculate the total mean squared loss $L = L_d^2 + L_{bc}^2$, as given by Equation (13).
 - c. **Backpropagation:** Compute the gradients of the loss function with respect to each network parameter ($\frac{\partial L}{\partial \mathbf{P}}$).
 - d. **Parameter Update:** Update the network parameters using an optimizer: $\mathbf{P}_{n+1} = \mathbf{P}_n - \eta \frac{\partial L}{\partial \mathbf{P}_n}$, where η is the learning rate.
 - 5: **Convergence Check:** If $L \leq \epsilon$, or the maximum number of epochs has been reached, terminate the training process.
 - 6: **Testing and Evaluation:**
 - a. Feed the testing dataset into the trained network with the saved, optimized parameters.
 - b. The network's output on this dataset constitutes the numerical solution for the BVP.
 - c. Perform post-processing and analysis by comparing the obtained solutions with reference data from other numerical or analytical methods.
-

3 Numerical Experiments

This section presents the application of our proposed feedforward neural network method to two distinct boundary value problems: sixth-order linear and nonlinear ODEs. A key objective of these experiments is to demonstrate the point-wise convergence and high accuracy of the neural network's approximate solution. Since both problems lack a known analytical solution, the performance of our neural network solver is rigorously validated by comparing its output against exact solutions. The entire training, testing, and optimization procedure for the network was conducted on personal laptops. This computational power allowed for the efficient execution of complex network architectures and extensive training over more than 30,000 epochs, with each run completing in a matter of minutes. For all numerical experiments, a consistent learning rate of 0.001 was utilized. Extensive testing was performed to identify the optimal combination of activation functions and optimizers for each problem. The subsequent subsections detail the specific results obtained for the two test cases, providing a comprehensive evaluation of the framework's performance.

The core of our solver is a fully connected, feedforward neural network, commonly known as a *multi-layer perceptron*. The architecture and training hyperparameters are specifically

configured as follows:

- **Network Architecture:** The network consists of an input layer with a single neuron for the spatial coordinate x , one hidden layer containing 16 neurons to approximate the complexity of the solution, and an output layer with a single neuron that provides the final solution, $\hat{y}(x)$.
- **Activation Functions:** The hyperbolic tangent (\tanh) function is utilized as the activation for the hidden layer, as its smoothness is beneficial for approximating differential equations. The output neuron employs a linear activation function to allow for an unbounded solution space.
- **Initialization Strategy:** Network weights are initialized using the Glorot Normal distribution (also known as Xavier initialization), a standard practice for deep networks that helps prevent vanishing or exploding gradients. All bias terms are initialized to zero.
- **Optimization Algorithm:** The Adamax optimization algorithm is used to update the network parameters during training, with a constant learning rate set to 1×10^{-3} .
- **Physics-Informed Loss Function:** A custom loss function guides the training process by encoding the physics of the problem. This function is the sum of two components: the mean squared error of the ODE residual, and the mean squared error from the specified boundary conditions at $x = 0$ and $x = 1$.
- **Training Procedure:** The model is trained for a total of 13,000 epochs to iteratively minimize the composite loss function until a satisfactory level of convergence is achieved.

Example-1: In this example, we apply the proposed neural solver to approximate the solution for the following linear, sixth-order boundary value problem.

$$y^{(6)}(x) - y(x) = -6e^x, \quad \text{for } 0 < x < 1, \quad (14a)$$

$$y(0) = 0, \quad y^{(2)}(0) = -1, \quad y^{(4)}(0) = -3, \quad (14b)$$

$$y(1) = 0, \quad y^{(2)}(1) = -2e, \quad y^{(4)}(1) = -4e. \quad (14c)$$

The exact solution to the BVP is given by $y(x) = (1 - x)e^x$. The composite loss function \mathcal{L} used to train the neural network approximation, $\hat{y}(x)$, is defined as the sum of the differential equation loss (\mathcal{L}_d) and the boundary condition loss (\mathcal{L}_b).

$$\text{Differential Equation Loss: } \mathcal{L}_d = \frac{1}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} \left(\hat{y}^{(6)}(x_i) - \hat{y}(x_i) + 6e^{x_i} \right)^2$$

$$\begin{aligned} \text{Boundary Loss: } \mathcal{L}_b = & \left(\hat{y}(0) \right)^2 + \left(\hat{y}^{(2)}(0) + 1 \right)^2 + \left(\hat{y}^{(4)}(0) + 3 \right)^2 \\ & + \left(\hat{y}(1) \right)^2 + \left(\hat{y}^{(2)}(1) + 2e \right)^2 + \left(\hat{y}^{(4)}(1) + 4e \right)^2 \end{aligned}$$

$$\text{Total Loss: } \mathcal{L} = \mathcal{L}_d + \mathcal{L}_b.$$

The subsequent table presents a detailed comparison between the solution approximated by our neural network and the exact analytical solution. An analysis of the results reveals that our neural network's predictions demonstrate a higher degree of accuracy than the numerical

solutions reported in [27]. This underscores the enhanced performance and precision of our proposed method for this particular problem.

Table 1: Comparison of analytical and numerical solutions with absolute error.

x	Analytical	Numerical	Error
0.0	1.00000000	1.00000000	0.00000000
0.1	0.99465383	0.99458730	0.00006652
0.2	0.97712221	0.97701275	0.00010945
0.3	0.94490117	0.94476765	0.00013351
0.4	0.89509482	0.89495212	0.00014270
0.5	0.82436064	0.82422036	0.00014028
0.6	0.72884752	0.72871995	0.00012757
0.7	0.60412581	0.60401964	0.00010617
0.8	0.44510819	0.44503129	0.00007690
0.9	0.24596031	0.24591930	0.00004101
1.0	0.00000000	0.00000000	0.00000000

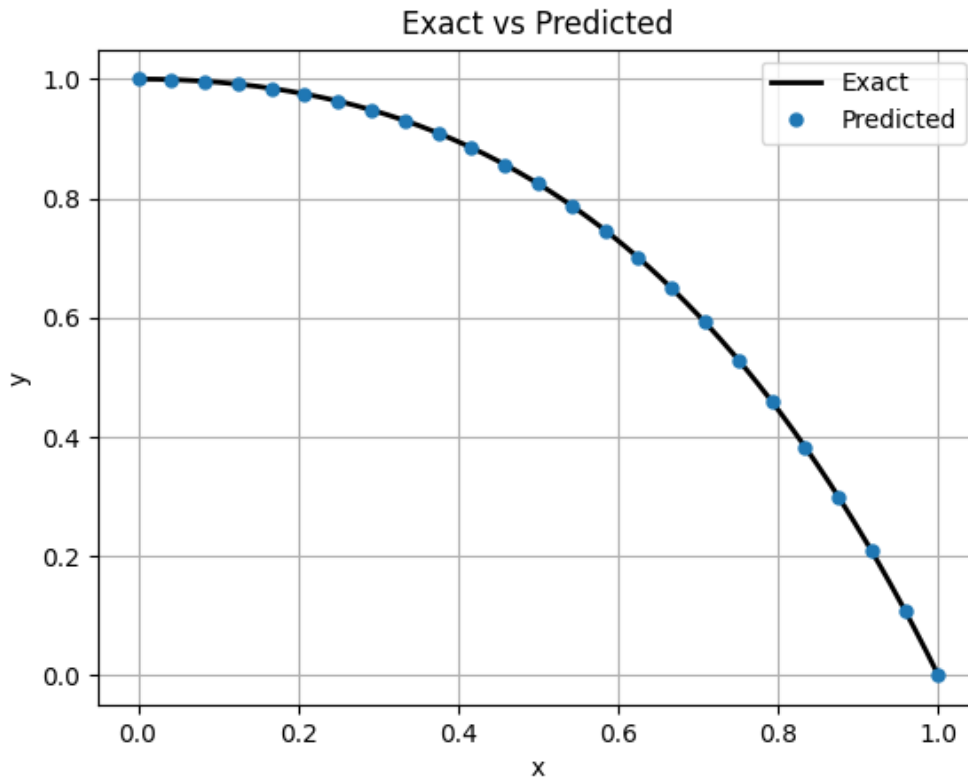


Figure 2: A graphical comparison between the exact solution (solid line) and the solution approximated by the neural network (markers). The close agreement between the two illustrates the high accuracy of our proposed model.

Example-2: In this example, we apply the proposed neural solver to approximate the solution for the following linear, sixth-order boundary value problem.

$$y^{(6)}(x) - e^{-x}y^2(x) = 0, \quad \text{for } 0 < x < 1, \quad (15a)$$

$$y(0) = 1, \quad y^{(2)}(0) = 1, \quad y^{(4)}(0) = 1, \quad (15b)$$

$$y(1) = e, \quad y^{(2)}(1) = e, \quad y^{(4)}(1) = e. \quad (15c)$$

The exact solution to the BVP is given by $y(x) = e^x$. The composite loss function \mathcal{L} used to train the neural network approximation, $\hat{y}(x)$, is defined as the sum of the differential equation loss (\mathcal{L}_d) and the boundary condition loss (\mathcal{L}_b).

$$\text{Differential Equation Loss: } \mathcal{L}_d = \frac{1}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} \left(\hat{y}^{(6)}(x_i) - e^{-x_i} \hat{y}^2(x_i) \right)^2$$

$$\begin{aligned} \text{Boundary Loss: } \mathcal{L}_b = & \left(\hat{y}(0) - 1 \right)^2 + \left(\hat{y}^{(2)}(0) - 1 \right)^2 + \left(\hat{y}^{(4)}(0) - 1 \right)^2 \\ & + \left(\hat{y}(1) - e \right)^2 + \left(\hat{y}^{(2)}(1) - e \right)^2 + \left(\hat{y}^{(4)}(1) - e \right)^2 \end{aligned}$$

$$\text{Total Loss: } \mathcal{L} = \mathcal{L}_d + \mathcal{L}_b.$$

The subsequent table presents a detailed comparison between the solution approximated by our neural network and the exact analytical solution. An analysis of the results reveals that our neural network's predictions demonstrate a higher degree of accuracy than the numerical solutions reported in [27]. This underscores the enhanced performance and precision of our proposed method for this particular problem.

Table 2: Comparison of the analytical solution, the Artificial Neural Network (ANN) solution, and the absolute error.

x	Analytical Solution	ANN Solution	Error
0.0	1.00000000	1.00821758	0.00821758
0.1	1.10517092	1.11291075	0.00773983
0.2	1.22140276	1.22851976	0.00711700
0.3	1.34985881	1.35623278	0.00637398
0.4	1.49182470	1.49736021	0.00553551
0.5	1.64872127	1.65334737	0.00462610
0.6	1.82211880	1.82578868	0.00366988
0.7	2.01375271	2.01644319	0.00269048
0.8	2.22554093	2.22725196	0.00171103
0.9	2.45960311	2.46035723	0.00075411
1.0	2.71828183	2.71812365	0.00015818

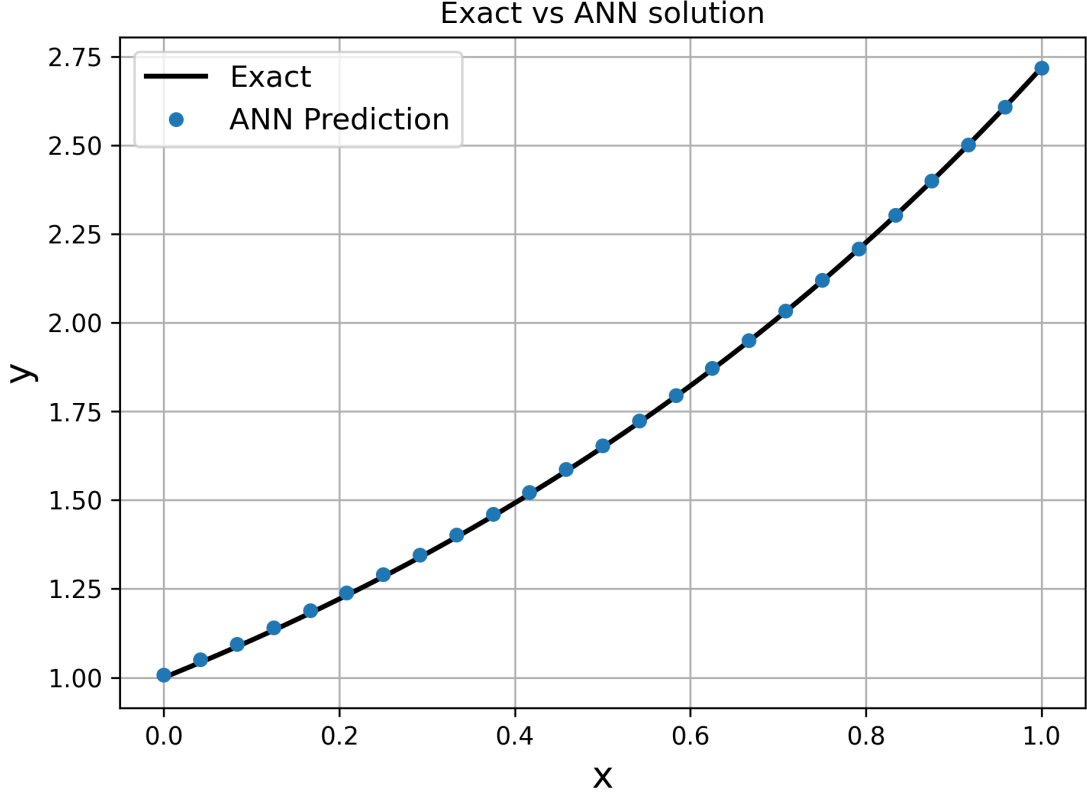


Figure 3: A graphical comparison between the exact solution (solid line) and the solution approximated by the neural network (markers). The close agreement between the two illustrates the high accuracy of our proposed model.

4 Conclusions

In this paper, we introduced a deep neural network framework as a powerful tool for approximating the solutions of high-order ordinary differential equations. Our approach is built on a straightforward and easy-to-implement physics-informed methodology. A single mean-squared error loss function effectively unifies the differential equation’s structure and its boundary conditions, which simplifies the problem-solving process. We also conducted a systematic investigation into the effects of various hyperparameters, such as activation functions, optimizers, and network architecture, to optimize the model’s performance. The efficacy of our framework was demonstrated on two challenging sixth-order boundary value problems: one linear and one nonlinear. In both cases, the neural network yielded solutions with a high degree of accuracy, closely matching the exact or established benchmark solutions. These results validate our method’s capability to handle the complexity and stiffness often associated with high-order differential equations without requiring problem-specific modifications.

Our findings suggest that this neural network approach is a compelling and robust alternative to classical numerical solvers like the Runge-Kutta or finite difference methods. A key advantage of our method is its simplicity; it circumvents the need for complex grid generation, iterative adjustments for boundary conditions, or linearization techniques for nonlinear problems. The solution is found directly by minimizing a well-defined objective function.

We plan to expand this framework to tackle various complex problems, such as systems involving fractional-order derivatives and partial differential equations. Additionally, establishing a more rigorous theoretical foundation for this approach is a crucial next step. Future inves-

tigations will focus on proving the convergence of the ANN method and formally defining the relationship between the loss function and the true solution error.

Acknowledgements

The author is deeply grateful to Dr. S. M. Mallikarjunaiah (Associate Professor of Mathematics, Texas A&M University–Corpus Christi) for proposing this research problem and for his invaluable guidance throughout the development of the methodology and neural network code. The valuable assistance of Ms. Pavithra Venkatachalapathy (PhD student, Texas Tech University) is also gratefully acknowledged.

References

- [1] Yoon, H., Mallikarjunaiah, S.. A finite element discretization of some boundary value problems for nonlinear strain-limiting elastic bodies. *Mathematics and Mechanics of Solids* 2022;27(2):281–307. URL: <https://doi.org/10.1177/10812865211020789>.
- [2] Yoon, H.C., Vasudeva, K.K., Mallikarjunaiah, S.M.. Finite element model for a coupled thermo-mechanical system in nonlinear strain-limiting thermoelastic body. *Communications in Nonlinear Science and Numerical Simulation* 2022;:106262.
- [3] Gou, K., Mallikarjuna, M., Rajagopal, K., Walton, J.. Modeling fracture in the context of a strain-limiting theory of elasticity: a single plane-strain crack. *International Journal of Engineering Science* 2015;88:73–82.
- [4] Mallikarjunaiah, S., Walton, J.. On the direct numerical simulation of plane-strain fracture in a class of strain-limiting anisotropic elastic bodies. *International Journal of Fracture* 2015;192(2):217–232.
- [5] Ferguson, L.A., Muddamallappa, M., Walton, J.R.. Numerical simulation of mode-iii fracture incorporating interfacial mechanics. *International Journal of Fracture* 2015;192(1):47–56.
- [6] Hornik, K., Stinchcombe, M., White, H.. Multilayer feedforward networks are universal approximators. *Neural networks* 1989;2(5):359–366.
- [7] Cybenko, G.. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 1989;2(4):303–314.
- [8] Lee, H., Kang, I.S.. Neural algorithm for solving differential equations. *Journal of Computational Physics* 1990;91(1):110–131.
- [9] Meade Jr, A.J., Fernandez, A.A.. The numerical solution of linear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling* 1994;19(12):1–25.
- [10] Logovski, A.C.. Methods for solving of differential equations in neural basis. In: [Proceedings] 1992 RNNS/IEEE Symposium on Neuroinformatics and Neurocomputers. IEEE; 1992, p. 919–927.
- [11] Lagaris, I.E., Likas, A., Fotiadis, D.I.. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks* 1998;9(5):987–1000.

- [12] Ngom, M., Marin, O.. Fourier neural networks as function approximators and differential equation solvers. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 2021;14(6):647–661.
- [13] Lau, L.L., Werth, D.. Oden: A framework to solve ordinary differential equations using artificial neural networks. *arXiv preprint arXiv:200514090* 2020;.
- [14] Rao, C., Sun, H., Liu, Y.. Physics-informed deep learning for computational elastodynamics without labeled data. *Journal of Engineering Mechanics* 2021;147(8):04021043.
- [15] Shi, E., Xu, C.. A comparative investigation of neural networks in solving differential equations. *Journal of Algorithms & Computational Technology* 2021;15:1748302621998605.
- [16] Dockhorn, T.. A discussion on solving partial differential equations using neural networks. *arXiv preprint arXiv:190407200* 2019;.
- [17] Gin, C.R., Shea, D.E., Brunton, S.L., Kutz, J.N.. Deepgreen: deep learning of green’s functions for nonlinear boundary value problems. *Scientific reports* 2021;11(1):1–14.
- [18] Terra, J.. Keras vs tensorflow vs pytorch: Understanding the most popular deep learning frameworks. 2021.
- [19] Mallikarjunaiah, S.M.. A deep learning feed-forward neural network framework for the solutions to singularly perturbed delay differential equations. *Applied Soft Computing* 2023;148:110863.
- [20] Venkatachalapathy, P., Mallikarjunaiah, S.M.. A feedforward neural network framework for approximating the solutions to nonlinear ordinary differential equations. *Neural Computing and Applications* 2022;;1–13.
- [21] Venkatachalapathy, P., Mallikarjunaiah, S.M.. A deep learning neural network framework for solving singular nonlinear ordinary differential equations. *International Journal of Applied and Computational Mathematics* ???;9.
- [22] Martinez, M., Rao, B.V.S.N., Mallikarjunaiah, S.M.. Approximation of one-dimensional darcy–brinkman–forchheimer model by physics informed deep learning feedforward artificial neural network and finite element methods: a comparative study. *International Journal of Applied and Computational Mathematics* 2024;10(3):102.
- [23] Goodfellow, I., Bengio, Y., Courville, A.. *Deep learning*. MIT press; 2016.
- [24] Glorot, X., Bengio, Y.. Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings*; 2010, p. 249–256.
- [25] McCulloch, W.S., Pitts, W.. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 1943;5(4):115–133.
- [26] Rumelhart, D.E., Durbin, R., Golden, R., Chauvin, Y.. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications* 1995;;1–34.
- [27] Wazwaz, A.M.. The numerical solution of sixth-order boundary value problems by the modified decomposition method. *Applied Mathematics and computation* 2001;118(2-3):311–325.