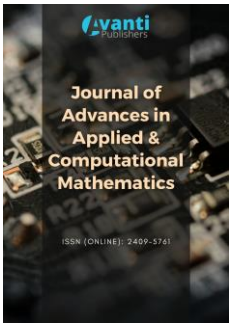
Published by Avanti Publishers

Journal of Advances in Applied & Computational Mathematics

ISSN (online): 2409-5761



A Gentle Introduction to Physics-Informed Neural Networks, with Applications in Static Rod and Beam Problems

Dimitrios Katsikis, Aliki D. Muradova and Georgios E. Stavroulakis*

Technical University of Crete, School of Production Engineering and Management, Computational Mechanics and Optimization Laboratory (www.comeco.tuc.gr), GR-73100 Chania, Greece

ARTICLE INFO

Article Type: Review Article

Keywords:
Differential equation
Python programming
Boundary value problem
Computational algorithm
Physics-informed neural networks

Timeline:
Received: March 30, 2022
Accepted: May 10, 2022
Published: May 31, 2022

Citation: Katsikis D, Muradova AD, Stavroulakis GE. A Gentle Introduction to Physics Informed Neural Networks, with Applications in Static Rod and Beam Problems. J Adv App Comput Math. 2022; 9: 103-128.

DOI: https://doi.org/10.15377/2409-5761.2022.09.8

ABSTRACT

A modern approach to solving mathematical models involving differential equations, the so-called Physics-Informed Neural Network (PINN), is based on the techniques which include the use of artificial neural networks and the method of fitting the governing differential equations at collocation points. In this paper, training of the PINN with an application of optimization techniques is performed on simple one-dimensional mechanical problems of elasticity, namely rods and beams. Different boundary conditions are considered.

Required computer algorithms are implemented using Python programming packages with the intention of creating neural networks. Numerical results are presented, and the efficiency of the proposed technique is investigated through numerical experiments with different numbers of epochs, batches, hidden layers, neurons, and collocation points.

The paper provides useful skills for using a PINN for different problems of solid mechanics. The proposed methodology is a continuation of our intention of using PINNs for problems of the theory of elasticity. The objectives are to present simply the main steps of constructing PINN and an implementation of it. A detailed explanation of the Python programming code, based on the scientific software Tensorflow, built in the Keras library and optimizers, may help compose an effective code for complicated models in mechanics.

PINNs are proposed in many recent publications to solve complicated direct and inverse problems. It seems to be a promising method that will play a central role in the development of computational mechanics in the near future. Nevertheless, the lack of educational material does not help new users to enter this scientific area. The present contribution describes the method for the solution of elementary rod and beam problems and gives computer codes that may help the reader to understand the method and to apply it to other problems.

*Corresponding Author
Email: gestavr@dpem.tuc.gr
Tel: 0030 6977308828

1. Introduction

In contrast with usual applications of a neural network (NN), based on giving examples of inputs and outputs in order to allow for training, a physics-informed neural network (PINN) exploits the input data and currently available techniques of automatic differentiation in order to create approximations of all terms involved in the differential equations and the initial/boundary conditions of the investigated model at every point in space and possibly in time.

Artificial neural networks (ANNs) were developed a long time ago (see, e.g., McCulloch and Pitts [16]). Artificial intelligence is a branch of computer science that seeks to create machines - models that operate autonomously in complex and changing environments (Russel and Norvig [25].

In recent years thanks to the technological progress in the development of robust, powerful computer software, an elaboration of backpropagation training and deep learning became possible to solve many complicated problems in science and engineering using artificial intelligence ([23]). The crucial role belongs to artificial neural networks. The history starts from the first hypotheses of Aristotle, who tried to explain a complicated mechanism of working of a human brain which receives and trains information coming from outside. The modern ANNs are based on the principle of operation of biological neurons.

The structure of ANN includes three main parts. The first part is intended for input data, and it is called an input layer. The second part consists of hidden layer/layers, where the treatment of the input data is performed with the help of activation functions for an approximate presentation of a solution to a considered problem. The third part is an output layer where the output data are produced.

ANNs have been applied in different areas of science and engineering. In particular, ANNs are employed in elastoplastic and contact problems in mechanics by minimizing energy. The Hopfield and Tank neural networks have been proposed by Kortesis and Panagiotopoulos [10] and Avdelas *et al*. [1]. The Feedforward NNs trained by the backpropagation algorithm has been used for an approximation of several problems in mechanics based on examples (supervised learning). Inverse and parameter-identification problems in mechanics have been solved by using backpropagation neural networks in Stavroulakis *et al*. [29], Stavroulakis [28], Waszczyszyn and Ziemiański [31]. Buckling loads in nonlinear problems for elastic plates have been calculated in Muradova and Stavroulakis [18]. A recent review of classical usage of neural networks within computational mechanics can be found in Yagawa and Oishi [32].

A recent direction in ANNs for solving scientific and mechanical problems, which became widely distributed and popular in modern times, is a physics-informed neural network. PINNs are used in case of lack of output data for training or/and complexity of the explored system and presence of a mathematical model, which is a differential equation or a system of differential equations with initial/boundary conditions. Recent developments related to Automatic Differentiation in neural networks in order to approximate the required derivatives and then an implementation with an application of open-source software have also promoted the development of PINNs (see, among others, Raisi *et al*. [22], Baydin *et al*. [2], Shin *et al*. [27], Karniadakis *et al*. [7]). The technique of using the governing partial differential equations, together with boundary conditions for training an artificial neural network to solve the problem, has been proposed by Lagaris *et al*. [11]. In a PINN, the input data are collocation points for fitting the governing differential equations. Here we can also mention the works of Tartakovsky *et al*. [30], Kadeethum *et al*. [6], Guo and Haghighat [5] and Zhang *et al*. [33].

Two main classes of problems: data-driven solution and data-driven discovery of partial differential equations are solved using PINN in the work of Raissi *et al*. [22]. Depending on the nature and arrangement of the available data, we devise two distinct algorithms, namely continuous-time and discrete-time models.

Another recent direction is the application of neural operators. Here we can mention the papers of Lu *et al*. [14] and Lu *et al*. [15] about the DeepONet neural operator and the works of Li *et al*. [13] and Kovachki *et al*. [9] with Fourier Neural Operator (FNO). These techniques are more general in the sense that they learn how to approximate the differential equations, and then they can solve any new problem with different initial or boundary conditions and physical parameters. The DeepONet has a NN for encoding the discrete input function

space and a NN for encoding the domain of the output functions, and it is based on the universal approximation theorem. In the FNO approach, the integral kernel is parameterized in the Fourier space.

In the work of Zhao *et al*. [34], a physics-informed convolutional neural network for the thermal simulation surrogate models is developed. This new approach is used for temperature field prediction of heat source layout without labeled data. In general, the method is important for engineering analysis and optimization.

A PINN architecture requires the classical elements of feedforward and backpropagation of neural networks, like nodes, hidden layers, and activation functions. Training of the neural network is performed with the use of optimization techniques. The PINNs are based on the universal approximation theorem for the underlying neural networks, and in this sense, they are considered suitable for the solution of problems with expected continuity, as the examples used here. If several parts with different material constants or cross-sections are considered, one may need to use families of different neural networks, one for each part, and add the continuity assumption as an additional term in calculating the error function.

In the present paper, the PINN is applied to beam and rod engineering problems. The simply supported, clamped, and free boundary conditions are considered. The programming code has been composed in Python 3.9 using Tensorflow 2.4.1 in the Pycharm environment. It is a modification of the code which implements the PINNs method for the problem of the Burgers equation in the paper of Raissi *et al*. [21]. The Adam's optimization algorithm (Kingma and Ba [8]), based on the gradient descent method and more commonly used in deep learning, together with the L-BFGS (Limited- Broyden–Fletcher–Goldfarb–Shanno) technique (Fletcher [4]) are applied in this work for solving the unconstrained nonlinear optimization problem arising during parameter estimation in training.

The paper is educational and aims to help people entering the area of PINN understand the method, based on elementary material from technical mechanics (rod and beam problems) and elementary material from neural networks (backpropagation neural network). Further knowledge is not required, provided that one uses the proposed software tools in Python. The paper provides useful skills for the understanding of a PINN with a detailed explanation of the application steps of it to rod and beam problems.

Here, the output of the PINN is compared with the exact solution to the problems. The effectiveness of neural networks in several areas gives hope that the method will contribute similarly to the solution of direct and, most importantly, inverse problems in mechanics. PINNs are directly comparable to collocation type, mesh-free numerical methods. Also, PINNs provide flexibility in the complexity of the domain of the definitions of the differential equations and initial/boundary conditions. For example, in variational methods, a special choice of local or global basis functions is crucial. In contrast, in PINNs, an approximation is represented with the help of activation functions, which are known beforehand. It is only needed to select an activation function for the considered problem properly. However, in order to reach a good approximation by PINN, sometimes many training epochs must be performed. Contemporary programming software intended for NNs allows us to overcome difficulties connected with the time of computations. Thus, PINNs can be applied to various problems in mechanics, science, and engineering. The PINN method is implemented and compared with the spectral method on an example of the Kirchhoff plate problem in [19]. In general, the question of comparing PINN with other numerical methods is still open and cannot be replied to easily. The effectiveness of using neural networks in several areas gives hope that the method will contribute similarly to the solution of direct and, most importantly, inverse problems in mechanics. PINN is directly comparable to collocation type, mesh-free numerical methods.

The present paper is organized as follows. In Section 2, elements of feedforward and backpropagation neural networks are given. The mathematical models involving partial differential equations and initial or boundary conditions for rods and beams are summarized in Section 3. Numerical examples are presented in Section 4. Conclusions are given in Section 5.

2. The Architecture of an Artificial Neural Network

The principle of operation of the biological neural networks of living organisms has played an important role in creating artificial neural networks. Through the biological NN understanding came the inspiration for machine

learning. Information processing and the way how biological neurons send signals is significantly more complex than the way how an artificial neural network works. Nevertheless, it becomes obvious that the human nervous system, and at a lower level of all living beings, consists of input units, central units for processing information, and output units. An interesting conclusion is that a collection of individual cells can lead to thought, action, and consciousness; in other words, the brain creates intellect (Searle [26]).

In accordance with biological NNs, the basic building block of artificial neural networks consists of a neuron that essentially receives information from network inputs or other neurons, processes it, and then transmits it to the subsequent neurons.

Let the values $x_1, x_2, \dots, x_n$ be the inputs for a neuron. Then for each neuron in an input layer of a neural network, the following quantity is calculated,

$$a = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \sum_{i=1}^{n} w_i x_i + b, \tag{2.1}$$

where $w_1, w_2, \dots, w_n$ are the weights, b is the bias, and n is the number of inputs. For convenience, the expression (2.1) can be written in the following form

$$a = \mathbf{w} \cdot \mathbf{x} + b, \tag{2.2}$$

where $\mathbf{w} = [w_1, w_2, \dots, w_n]$, $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$.

Furthermore, we introduce an activation function that is computed for the a**.** An activation function plays an important role in ANNs. It decides whether a neuron should be activated or not, which depends on the importance of the neuron's input in the process of prediction. The activation functions were introduced based on harmonic analysis (e.g., Cande`s [3]). The most used activation functions are the Heaviside, Sigmoid /logistic, Hyperbolic tangent (tanh), and ReLU. For example, the sigmoid nonlinear function is

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

and the tanh nonlinear function is

$$f(x) = \tanh(\mathrm{x}) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.4}$$

The sigmoid function is used in the logistic regression classification algorithm. It shows very smooth behavior. The hyperbolic tangent (tanh) is also a smooth nonlinear activation function applied in this work for the examples in Section 4.

The connectivity of many neurons like those described above contributes to an ANN. The procedure which supplies the network with the information spreading from the input layer to the outputs, penetrating all the hidden layers with neurons, is called feedforward. An example of a neural network is depicted in Figure **2.1**.

In order to explain the feedforward way of ANN, it is useful to define the following weight matrices. Let a matrix W_1 contains the weights between the connections of the neurons from the input layer to the 1st hidden layer, a matrix W_2 contains the weights from the 1st hidden layer to the 2nd hidden layer and a matrix W_3 includes the weights from the 2nd hidden layer to the output layer (Figure **2.1**). In general, the elements of the matrices W_k are denoted by $w_{k,ij}$, where $w_{k,ij}$ is the weight from i - neuron of layer $k-1$ to the j - neuron of layer k. By convention, the layers are numbered as $k = 1, 2, \dots, N_L$, where N_L is the number of layers. The input layer is numbered as 0, and the corresponding inputs in it with $i = 1,2, \dots, n_0$ $(n_0 = n)$. For the elements $w_{1,ij}$ of W_1, $i = 1,2, \dots, n_0$, $j = 1,2, \dots n_1$, for the elements $w_{2,ij}$ of W_2, $i = 1,2, \dots n_1$, $j = 1,2, \dots n_2$ and so on. The quantity n_k refers to the number of neurons in layer k. The output of each layer is a vector denoted by $\mathbf{z}_k$ and therefore, the input vector can be written as $\mathbf{z}_0$. Figure **2.1** applies $N_L = 3, n_0 = 4, n_1 = n_2 = 5, n_3 = 3$. Hence, the dimensions of matrices W_1, W_2, W_3 are derived as (4×5), (5×5), and (5×3), respectively.

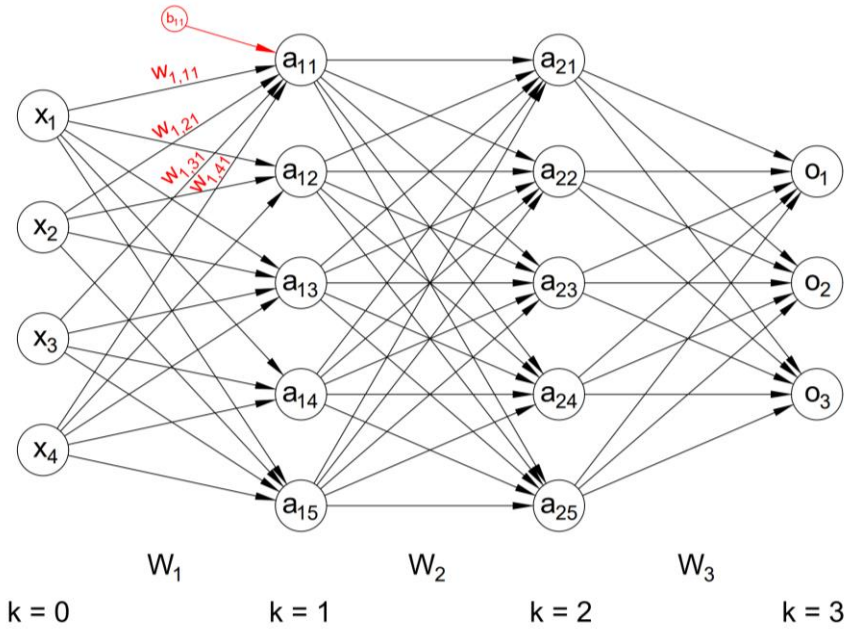


Figure 2.1*:* ANN with 4 inputs, 2 hidden layers with 5 neurons, and 3 outputs. Weights of connections leading to node a_{11} are also depicted.

An additional vector $\mathbf{b}$ includes the bias of the neurons. The input layer has no bias, so the numbering of the vectors starts with $\mathbf{b}_1$ which is the vector that includes the biases of the 1st hidden layer. The elements of the vectors of biases $\mathbf{b}_k$ are defined as b_{kj}.

A general formula for the feedforward process with the activation function $\sigma(x)$ is

$$\begin{aligned} \mathbf{z}_0 &= \mathbf{x}, \\ \mathbf{z}_k &= \sigma(W_k^T \cdot \mathbf{z}_{k-1} + \mathbf{b}_k), \qquad k = 1,2,\dots,N_L. \end{aligned} \tag{2.6}$$

where W_k is the matrix of the dimension $(n_{k-1} \times n_k)$ and $\mathbf{z}_{N_l}$ is the output of the neural network.

Tensorflow uses the equation (2.6) repeatedly for all layers to calculate the output of the AAN for every input from the data set.

The training of a network is perhaps the most important part of machine learning since, through this process, a loss function (prediction error of neural network) is minimized, and the weights and biases are updated. To minimize the loss function, it is necessary to calculate the derivatives of the function with respect to its variables, weights, and biases. The backpropagation algorithm calculates all the necessary derivatives, with the rule of chain derivation, starting from the last layer and moving backward to the input layer. The general steps of the training process are the following:

1. Forward Pass - Input the data for the artificial neural network, a number of epochs, and physical parameters for the problem.
2. Loss function calculation.
3. Calculation of some derivatives of the Loss function in relation to all trainable parameters starting from the last layer and continuing backward (backpropagation).
4. Change the trainable parameters in the direction that minimizes the Loss function.
5. Repeat steps 1, 2, 3 and 4 for all data (training set).

Repeat steps 1, 2, 3, and 4 for those training courses (epochs) that have been defined.

In order to complete a training cycle, it is necessary to calculate the loss function and change the parameters for each separate pair of inputs-outputs, which causes a high computational cost and long delays in the convergence of the algorithm. For this reason, often in applications, a training set is divided into individual sections called batches.

The main optimization algorithms used in ANN training are the Gradient Decent Algorithm, the L-BFGS algorithm, and the Adam (Adaptive moment estimation) algorithm. The Gradient Descent Algorithm minimizes the loss function with respect to the weights and the biases of the network, which change in the opposite direction of the feedforward. The partial derivatives of the function for each parameter are calculated (Ruder [24]). The L-BFGS algorithm opens up to the family of quasi-Newton methods developed by Broyden, Fletcher, Goldfard, and Shanno. Newton's method for minimization problems requires the calculation of the Hessian matrix of the objective function, i.e., it requires the calculation of second derivatives with respect to all the parameters, which is computationally prohibitive in deep learning problems. In contrast, the quasi-Newton methods approach finds zeroes or local maxima and minima of functions. The Adam algorithm, proposed by Kingma and Lei Ba [8], has the advantage because it calculates a learning rate separately for different parameters.

3. Implementation of Physics Informed Neural Network

The first proposals for the use of ANN for solving differential equations have been given in the works of Lee and Kang [12], Lagaris *et al*. [11], and Meade Jr and Fernadez [17]. Here one can also mention the most recent work of Karniadakis *et al*. [7] on this topic and the work of Muradova and Stavroulakis [19], where the PINN for solving the Kirchhoff plate bending problem is applied. Recent developments of the PINNs method are presented in Raissi *et al*. [21] and Peng *et al*. [20], in which a reader can find more detailed information on the implementation of the method.

Most engineering problems are modeled through differential equations. One way to solve them is to find an analytical solution, i.e., the expression of unknown function $u(x)$, which satisfies exactly the differential equations and the initial/boundary conditions. Another way is to use semi-analytical or numerical methods, such as, e.g., the Runge – Kutta methods or finite element method (FEM) etc., in which the $u(x)$ is approached point by point by an approximate solution, function $\hat{u}(x)$. The physics-informed neural network method is a numerical method intended to find an approximation of the exact solution $u(x)$ via an ANN. Therefore, $u(x) \approx \hat{u}(x) = NN\ (x)$.

The phrase 'Physics-Informed' refers to the fact that the network is trained based on the mathematical model imposed by physics and mathematics. It is a kind of self-supervised learning where the training is carried out with pairs of inputs and desired outputs that are not introduced before but are produced by differential equations themselves. The network inputs are only one independent variable if the model consists of static ordinary differential equations and two or more independent variables of the function if the model consists of partial differential equations. The inputs are derived from the domain of the definition of $u(x)$. The corresponding output during feedforward supplies the function $NN(x)$. The peculiarity of the method and the point that distinguishes it from the common ANNs is the fact that during the training, in addition to the calculation of the derivatives of the loss function for all trainable parameters of the network, one uses the derivatives of the neural network with respect to input variables. If it is a solution of a partial differential equation, all partial derivatives of $\hat{u}(x)$ concerning the independent variables are calculated. A loss function is constructed in order to check whether the approximate solution $\hat{u}(x)$ is far from $u(x)$.

3.1. Feedforward in a PINN

This section studies the process of propagation of input data from the input layer to the hidden layers and then to the output of the neural network. This process will be examined through an example of the second-order ordinary differential equation with initial conditions,

$$\frac{d^2u(x)}{dx^2} + a\,\frac{du(x)}{dx} = b, \qquad x \in (x_0, L], \qquad L \in R, \tag{3.1}$$

$$u(x_0) = u_0, \quad \frac{du(x_0)}{dx} = u_1, \tag{3.2}$$

where *a, b* are the constant parameters. To find the solution $u(x)$ of the problem (3.1), (3.2), a PINN is used, where the output of the neural network, $NN(\mathrm{x})$, is the approximation of the solution $u(\mathrm{x})$. The following formulas describe the calculation of the output vector from each layer.

$$\begin{aligned} \mathbf{z}_0 &= \mathbf{x} \\ \mathbf{z}_k &= \sigma(W_k^T \cdot \mathbf{z}_{k-1} + \mathbf{b}_k), \qquad k = 1,2,\dots,N_L - 1. \\ \mathbf{z}_{N_L} &= W_{N_L}^T \cdot \mathbf{z}_{N_L-1} + \mathbf{b}_{N_L} \end{aligned} \tag{3.3}$$

The PINNs output is linearly dependent on the last hidden layer. Note that the output layer (3.3) does not enter the activation function as it usually is in ANNs (see (2.6), $k = N_L$). The z_{N_L-1} is defined as the vector containing the outputs of the last hidden layer, $W_{N_L}^T$ is the matrix containing the weights between the last hidden layer and the output of the network and $\mathbf{b}_{N_L}$ as the bias vector of the output layer. Thus, the approximate solution is written as

$$\hat{u}(\mathbf{x}) = NN(\mathbf{x}) = W_{N_L}^T \cdot \mathbf{z}_{N_L-1} + \mathbf{b}_{N_L}. \tag{3.4}$$

The elements of the input vector $\mathbf{x}$ are the collocation points for (3.1). Note that the neural network provides a value of the displacement at each collocation point, $NN(x_i),\ i = 1,2,\dots,n$.

3.2. Training & Backpropagation in PINNs

The training process is performed by modifying the trainable parameters (weights and biases) in such a way that the function $NN(x) = \hat{u}(x)$ satisfies the differential equation and the initial conditions with as little error as possible. The weights and biases are adapted by minimizing the loss function (mean square error),

$$MSR = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{d^2\hat{u}(x_i)}{dx^2} + a\ \frac{d\hat{u}(x_i)}{dx} - b\right)^2 + (\hat{u}(x_0) - u_0)^2 + \left(\frac{d\hat{u}(x_0)}{dx} - u_1\right)^2, \tag{3.5}$$

where $x_i, i = 1,2,\dots,n$ are the collocation points. For the construction of the loss function, the error from the differential equation and errors from initial – boundary conditions must be added. In Figure **3.1**, the architecture of the PINN for this example is presented.

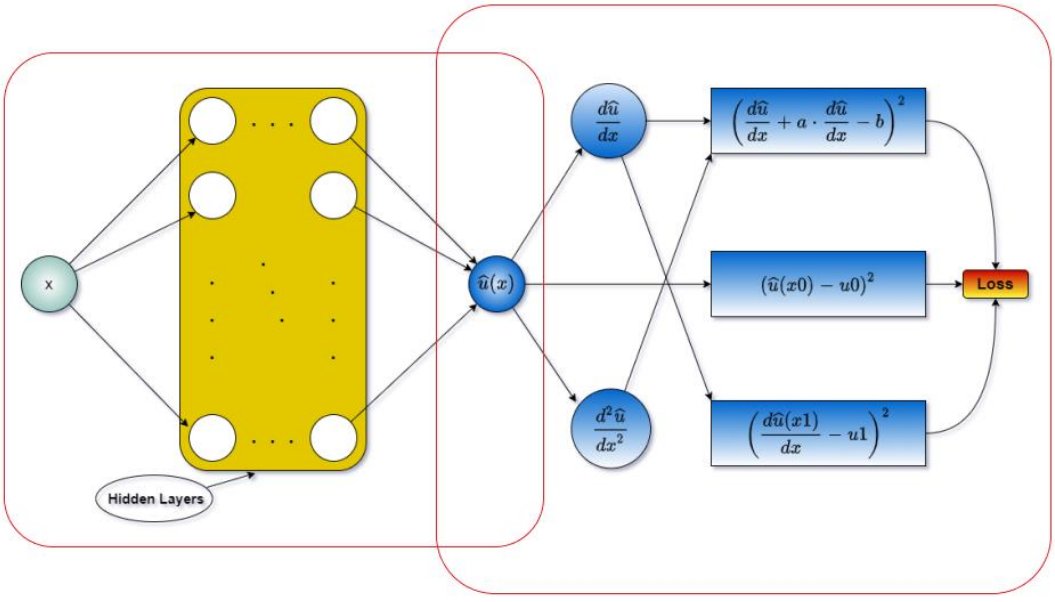


Figure 3.1: The architecture of ANN, in combination with the formation of the differential equation and the initial conditions. The first block is the classical backpropagation neural network. The second block is the extension for physics-informed learning.

The training algorithm can work with known optimization algorithms such as the Gradient Descent or L-BFGS, or Adam method. In this work, first L-BFGS has been used and then the Adam method. In applying the method to real problems listed below, the training set is divided into batches. The steps of implementation of the PINN in the general case are the following:

1. Input the data for the neural network: the collocation points for the differential equation and initial/boundary conditions, weights, biases, a number of epochs, and the physical parameters for the mechanical problem.
2. Calculate $\hat{u}(x)$ as a function of the values of x_i, using the activation function, weights, and biases (Feedforward).
3. Compute necessary derivatives of $\hat{u}(x)$ with respect to x contained in the differential equation and in the initial/boundary conditions.
4. Construct the loss function.
5. Calculate derivatives of the loss function concerning all trainable parameters starting from the last layer (Backpropagation).
6. Minimize the loss function using an optimization technique.
7. Update the weights and biases parameters in the NN.
8. Repeat steps 2, 3, 4, 5, 6, and 7 for all data (training set) that have been set until the desired accuracy for the loss function is achieved or until training cycles (epochs).

3.3. Automatic Differentiation

For the implementation of PINN, the calculation of all derivatives appearing in the differential equation, in the initial/boundary conditions, and for all trainable parameters is required. Therefore, an efficient way of calculating derivatives is needed so that the process is achieved faster with low memory requirements. The Automatic Differentiation using Python's Tensorflow library is implemented here. Tensorflow is an open-source program for machine learning and deep learning. It provides suitable functions which are used extensively in this work. The Automatic Differentiation option mainly combines two positive options. The chain rule essentially results with the same accuracy as the "symbolic differentiation" with comparable speed. Information on how it works has been taken from the article of Baydin *et al*. [2]. Below is an example used to illustrate the process of calculating derivatives with the help of Automatic Differentiation.

3.4. Implementation in Python Code Via Tensorflow

The implementation of Automatic Differentiation in Python is performed through Tensorflow, which includes a package of functions suitable for implementing ANNs. The Tensorflow provides the tf.GradientTape() API (Application Programming Interface) function which is responsible for calculating derivatives through Automatic Differentiation Reverse Mode. The following commands are used for the calculation of the derivative of the sigmoid function at a specific point x:

```
import TensorFlow as tf
import NumPy as np
x = tf.Variable(3.0)
with tf.GradientTape() as tape:

   y = tf.sigmoid(x)
dy_dx = tape.gradient(y, x)
print(f"dy_dx = {dy_dx}")
```

If, for example, there is a simple neural network without any hidden layer, with 3 input neurons and 2 output neurons, then the following program calculates the loss function and the Jacobian array that also includes some derivatives of the loss function with respect to trainable parameters.

```
w = tf.Variable([[0.5, 0.5], [0.5, 0.5], [0.5, 0.5]])
b = tf.Variable(tf.zeros(2, dtype=tf.float32), name='b')
x = [[1., 2., 3.]]
with tf.GradientTape(persistent=True) as tape:
z = x @ w + b    # z = [z1  z2]
loss = tf.reduce_mean(z ** 2)  # loss = z1^2 + z2^2
print("Loss = ")
print(loss)
[dL_dw, dL_db] = tape.gradient(loss, [w, b])
print("dL_dw = ")
print(dL_dw)
print("dL_db")
print(dL_db)
```

The command loss = tf.reduce_mean(z ** 2) creates the error function:

3.5. Construction of ANN Models through Keras Library

The Keras module library of Tensorflow allows us to implement the Automatic Differentiation steps described in the previous section. The Keras is a high-level interface of the Tensorflow 2 platform (API) to implement machine learning models and especially deep learning, which usually requires less computing time. Practical use of the Keras for the Automatic differentiation is shown in the following command lines:

```
import TensorFlow as tf
layer = tf.keras.layers.Dense(2, activation='relu')
x = tf.constant([[1., 2., 3.]])
with tf.GradientTape() as tape:
  # Forward pass
  y = layer(x)
  loss = tf.reduce_mean(y**2)
# Calculate gradients with respect to every trainable variable
grad = tape.gradient(loss, layer.trainable_variables)
# layer.trainable_variables is a list
print(layer.trainable_variables)
for var, g in zip(layer.trainable_variables, grad):
print(f'{var.name}, shape: {g.shape}')
```

4. Numerical Results and Discussions

4.1. Numerical Examples

In this section, the PINN for some engineering problems, in particular, for the models of solid deformable rods and beams, is tested. The considered problems include ordinary differential equations and initial and boundary conditions.

Problem 1: A rod with distributed forces (Figure **4.1**).

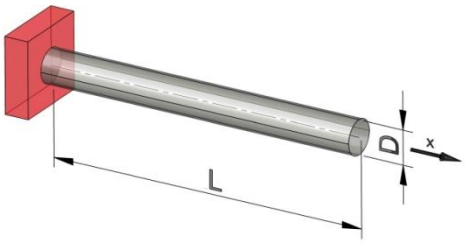


Figure 4.1: Rod fixed to one end.

If distributed tensile forces cx exert on a rod, then the load at an elementary point x is equivalent to $\int_x^L c\,x\,dx = \frac{c}{2}(L^2 - x^2)$ where L is the total length of the rod, c is the constant coefficient. The stresses on the rod are computed as

$$\sigma = \frac{Load}{Area} = \frac{c\,(L^2 - x^2)}{2\,A}, \tag{4.1}$$

where A is the cross-sectional area. In the rod subjected to axial loading, the strain is defined through the first derivative of the deformation (displacement) at a specific point

$$\varepsilon = \frac{du}{dx}. \tag{4.2}$$

Here $u(x)$ is the displacement of the rod at point x. Further, as it is known the Young's modulus (modulus of elasticity) is expressed as $E = \sigma/\varepsilon$. Therefore, it follows that,

$$E\,\frac{du}{dx} = \frac{c\,(L^2 - x^2)}{2\,A} \tag{4.3}$$

or

$$AE\,\frac{du}{dx} = \frac{c}{2}\,(L^2 - x^2). \tag{4.4}$$

Hence,

$$AE\frac{d^2u}{dx^2} = -cx, \quad x \in (0, L). \tag{4.5}$$

The bountry conditions for the fixed rod are

$$u(0) = 0, \quad \left.\frac{du}{dx}\right|_{x=\mathrm{L}} = 0. \tag{4.6}$$

The exact solution of (4.5), (4.6) is

$$u(x) = \frac{c}{6\,AE}(-x^3 + 3L^3x). \tag{4.7}$$

The PINN is also applied to the case when except for the tensile forces, an additional force of the opposite direction is applied to the end of the rod. The boundary conditions, in this case, are defined as follows:

$$u(0) = 0, \quad E\left.\frac{du}{dx}\right|_{x=L} n + \frac{cL^2}{A} = 0, \tag{4.8}$$

The exact solution for (4.5) and (4.8) is

$$u(x) = \frac{c}{AE}\left(-\frac{x^3}{6} + L^2 x\right). \tag{4.9}$$

The composed code consists of the main module and other 5 modules, the network, the layer, the PINN, the optimizer, and the tf_silent modules. The network module includes a class named Network which, through a function named build, constructs the network and returns it. As it is mentioned in Section 3, the output of the last hidden layer does not enter any activation function, and the approximate solution is calculated by the formula (3.4). This detail is confirmed and becomes even more evident in the build function in which, for the calculation of the output, no activation function is mentioned in contrast to the other layers. Below are the corresponding lines of code from the module network.

````
# input layer
inputs = tf.keras.layers.Input(shape=(num_inputs,))
# hidden layers
x = inputs
for layer in layers:
   x = tf.keras.layers.Dense(layer, activation=activation,
      kernel_initializer='he_normal')(x)
# output layer
outputs = tf.keras.layers.Dense(num_outputs,
   kernel_initializer='he_normal')(x)
return tf.keras.models.Model(inputs=inputs, outputs=outputs)
````

The module layer has the role of calculating the necessary derivatives of the network with respect to the input **x**, which in this case are the derivatives $d\hat{u}/dx$, $d^2\hat{u}/dx^2$. This module includes a class named GradientLayer () which is called by the module PINN and takes as input the network. Finally, the numerical values of the first and the second derivatives are returned as well as the numerical value of $\hat{u}$(x) for each x value that enters the network. The derivatives are calculated using the Tensorflow GradientTape() function. Below are the corresponding lines of code from the module layer.

```
with tf.GradientTape() as g:
   g.watch(x)
   with tf.GradientTape() as gg:
      gg.watch(x)
      u = self.model(x)
   du_dx = gg.gradient(u, x)
d2u_dx2 = g.gradient(du_dx, x)
return u, du_dx, d2u_dx2
```

The module PINN communicates with the module layer as it receives the derivatives from it in order to construct the second neural network called the PINN. It includes the class named PINN, in which the PINN network is constructed and finally returned. In fact, the inputs **x** that enter two networks are used to calculate the derivatives through the module layer, and then the differential equation and the initial conditions are constructed as an additional custom layer to complete the PINN network and are finally returned.

For every $x \in (0, L)$ it is calculated the quantity $u_1 = AE\frac{d^2\hat{u}}{dx^2} + cx$ for the equation, then the quantity $u_2 = \hat{u}(0)$ and the quantity $u_3 = E\frac{d\hat{u}}{dx}\Big|_{x=L}$ n $+ \frac{cL^2}{A}$ are calculated for the boundary conditions, respectively. The target values for

the network output are set to $u_1 = 0$, $u_2 = 0$ και $u_3 = 0$. Below are the code lines of the module PINN in which the GradientLayer class is called from the module layer, as well as the u_1, u_2, u_3 are calculated, respectively.

```
    self.grads = GradientLayer(self.network)

# compute gradients
u, du_dx, d2u_dx2 = self.grads(x_1)
# equation output being zero
u_1 = self.A * self.E * d2u_dx2 + self.c * x_1
# initial condition output
u_2 = self.network(x_2)
# boundary condition output
u, du_dx, d2u_dx2 = self.grads(x_3)
u_3 = self.E * du_dx * self.n + self.c * (self.L)**2 / self.A
# build the PINN model for BAR' equation
return tf.keras.models.Model(
   inputs=[x_1, x_2, x_3], outputs=[u_1, u_2, u_3])
```

The module optimizer performs the PINN network training process through the L-BFGS optimization algorithm. It includes the class, named L_BFGS_B, which takes as parameters the neural network PINN, the inputs **x** with the corresponding target values u_1, u_2, u_3, as well as the parameters factr, m, maxls, maxiter. Initially the weights are initialized through the function set_weights (self, flat_weights). Also, in this module, there is the function tf_evaluate (self, x, y), in which the derivatives of the loss function are calculated with respect to the common weights and the biases of the two networks. In the fit (self) function, the parameters are modified through the L-BFGS algorithm. Below are the code lines of the module optimizer in which the backpropagation process and the parameter modification process are performed, respectively.

```
with tf.GradientTape() as g:
   loss = tf.reduce_mean(tf.keras.losses.mse(self.model(x), y))
grads = g.gradient(loss, self.model.trainable_variables)
return loss, grads

scipy.optimize.fmin_l_bfgs_b(func=self.evaluate, x0=initial_weights,
   factr=self.factr, m=self.m, maxls=self.maxls, maxiter=self.maxiter)
```

The tf_silent module is auxiliary, and its only task is to delete the unnecessary messages that are printed at the start of the program due to the call of the Tensorflow library.

The main module must first include the libraries that will be used, as well as the necessary classes from the modules mentioned above. Specifically, in the main program the PINN class from the module PINN, the Network class from the module network, and finally, the L_BFGS_B class from the module optimizer are called. From the line containing the if __name__ == '__main__' command: and below, the program starts running. Initially, the values of the variables num_train_samples and num_test_samples are defined, whereas in this case, the values 1000 and 100 were given, respectively. Num_train_samples is defined as the number of *x* points (colocation points) with which the training will be done, while num_test_samples is defined as the number of *x* points that after the training will be used to check and construct a graph of $\hat{u}(x)$. The physical constants A, E, c, and L are defined. Then the input and output vectors - target x_1, x_2, x_3 and u_1, u_2, u_3 are defined, respectively.

```
# build a core network model
network = Network.build()

# build a PINN model
pinn = PINN(network, E, A, c, n, L).build()

# create training input
x_1 = 2*L*np.random.rand(num_train_samples, 1)      # x_1 =  0 ~ 2*L
x_2 = np.zeros((num_train_samples, 1))              # x_2 =  0
x_3 = 2*L*np.ones((num_train_samples, 1))           # x_3 =  2*L
# create training output - target
u_1 = np.zeros((num_train_samples, 1))              # u_1 = 0
u_2 = np.zeros((num_train_samples, 1))              # u_2 = 0
u_3 = np.zeros((num_train_samples, 1))              # u_3 = 0

lbfgs = L_BFGS_B(model=pinn, x_train=x_train, y_train=y_train)
lbfgs.fit()
```

The two networks for Problem 1 are illustrated in Figure **4.2**.

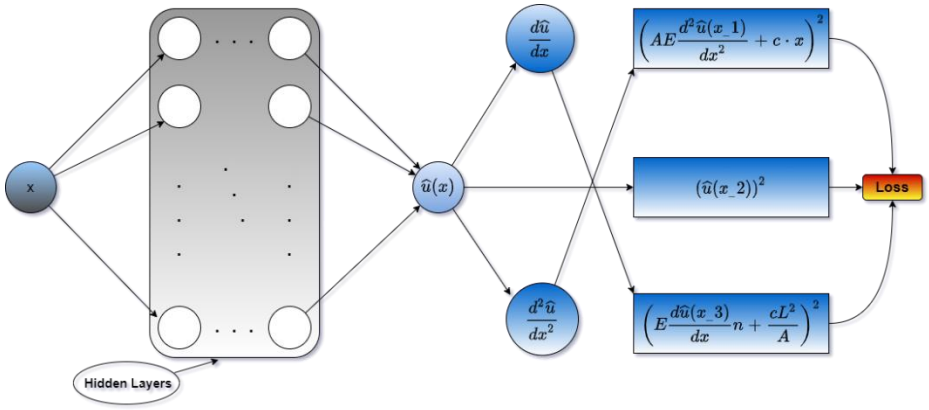


Figure 4.2*:* Illustration of the two networks for Problem 1**.**

The numerical results are shown in Figure **4.3** and Tables **4.1** and **4.2** after applying the L-BFGS and the Adam algorithms, respectively.

Looking at Table **4.1**, one first gets the impression that the original program with the L-BFGS optimizer brings very satisfactory approaches. Successful program executions were mediated by several failed approaches with an error greater than 0.05 or even worse. In these cases, the graph of the approximate solution was quite common with the graph of the analytical solution $u(x)$, but their difference in the graph was noticeable. The lack of stability became the reason for using the Adam algorithm as an optimizer, as well as the removal of the module network and module optimizer in order to make the code more readable and easier to manage.

The Adam algorithm appeared to have much better and more stable performance. As shown in Table **4.2**, the error did not show any significant improvement compared to the previous program, but during the execution, there was great stability.

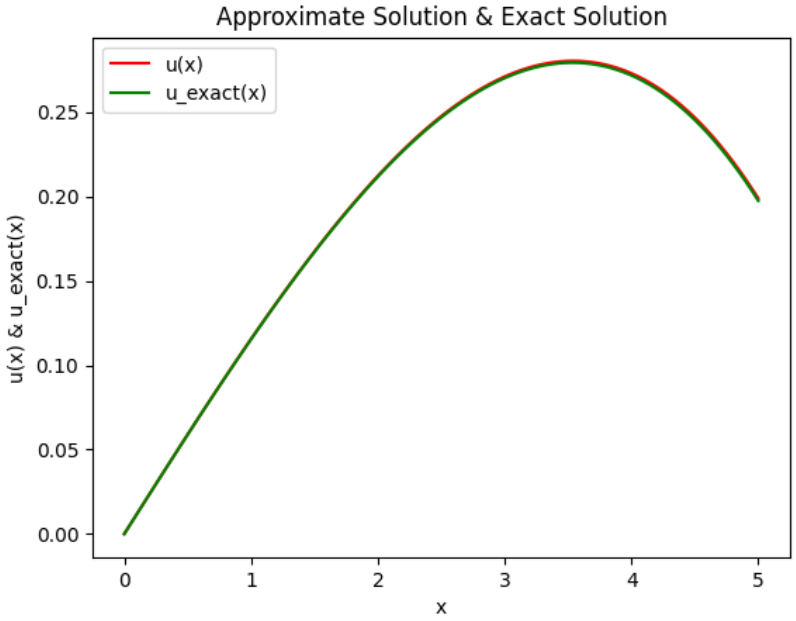


Figure 4.3*:* Graphical representation – No 8 from Table 4.1.

Table 4.1: The results of solving Problem 1 with using the L-BFGS algorithm.

No.	Layers	Training Sample	Loss Mse	Time(Sec)
1	[15, 30, 60, 30, 15]	1000	0.009804	7167
2	[15, 30, 60, 30, 15]	1000	0.002842	4508
3	[15, 30, 60, 30, 15]	1000	0.001222	388
4	[15, 30, 60, 30, 15]	1000	0.004778	5150
5	[15, 30, 60, 30, 15]	1000	0.000596	5341
6	[15, 30, 60, 30, 15]	1000	0.002389	1323
7	[15, 30, 60, 30, 15]	1000	0.003372	3119
8	[20, 30, 60, 40, 20]	1000	0.000352	3307
9	[20, 30, 60, 40, 20]	1000	0.001479	3060

Table 4.2: The results of solving Problem 1 using Adam's algorithm.

No.	Layers	Batch Size	Epochs	Training Sample	Loss Mse	Time(Sec)
1	[15, 30, 60, 30, 15]	16	1151	1000	0.008200	256
2	[30, 60, 30]	16	398	10000	0.008100	736
3	[30, 60, 30]	32	982	1000	0.007300	1015
4	[30, 60, 30]	32	1717	1000	0.009700	1798

Comparison with classical computational methods will be fair if, for example, within a finite element analysis, the same number of nodes with the collocation points of PINN is used. The results for this academic example are identical.

Problem 2. Simply Supported Beam (Figure **4.4**).

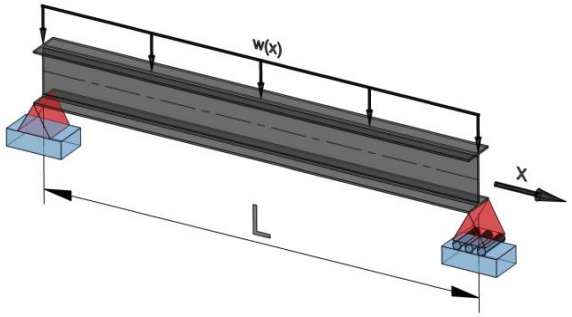


Figure 4.4: Simply supported beam.

If a bending moment M is exerted on a beam where $M = -Px$ with the concentrated load P, then from the static theory, it is known that the differential equation describing the beam in bending reads:

$$\frac{d^2y}{dx^2} = \frac{M(x)}{EI}. \tag{4.10}$$

However, if a distributed load $w(x)$ is applied to the beam understudy, then the differential equation (4.10) should be properly formed. For this to happen, the magnitude V is defined as the shear force and the following relationships hold for the magnitude,

$$\frac{dM}{dx} = V, \qquad \frac{dV}{dx} = -w. \tag{4.11}$$

Hence in virtue of (4.10) and (4.11), it follows

$$\frac{d^4y}{dx^4} = -\frac{w(x)}{EI}. \tag{4.12}$$

If it is a totally clamped beam, and the beam has length L, it holds that $y(0) = 0$, $M(0) = 0, y(L) = 0$, and $M(L) = 0$. Thus, the boundary conditions are

$$y(0) = 0, \quad y(L) = 0, \qquad \frac{d^2y(0)}{dx^2} = 0, \qquad \frac{d^2y(L)}{dx^2} = 0, \tag{4.13}$$

In order to have realistic values a W - Beam (Wide Flange Beam) is considered, with a measure of elasticity E = 200 GPa (200 * 1e9 Pa), moment of inertia I = 0.000038929334 (m^4), length L = 2.7 m and distributed load w = 60 KN / m.

The exact solution of the problem (4.6), (4.7) is

$$y(x) = \frac{w}{24\,EI}(-x^4 + 2Lx^3 - L^3x). \tag{4.14}$$

The programming code performed is similar to the code in Problem 1. However, the L-BFGS optimization algorithm was replaced by the Adam algorithm. In addition, some simplifications were made to make the code more readable, while the Class called StopAtLossValue (Callback) was added, which defines a minimum loss function value as the termination criterion, as well as a runtime counter so that there is an image of the program computational costs.

In the module PINN, through the function build, it is built a network u_model and network PINN_model. The vectors u_1, u_2, u_3 for the differential equation and boundary conditions are computed.

```
# compute gradients
u, d2u_dx2, d4u_dx4 = grads(x_1)
# equation output being zero
u_1 = d4u_dx4 + self.w/(self.E * self.I)
# initial condition output
u_2 = u_model(x_2)
u, d2u_dx2, d4u_dx4 = grads(x_3)
u_3 = d2u_dx2

# build the PINN model for beam's equation
return u_model, tf.keras.models.Model(
   inputs=[x_1, x_2, x_3], outputs=[u_1, u_2, u_3])
```

The module layer has the same structure as in the previous problem with the difference that in this case, the calculations until the 4th order derivatives of the u_model network is needed.

```
with tf.GradientTape() as g:
   g.watch(x)
   with tf.GradientTape() as gg:
      gg.watch(x)
      with tf.GradientTape() as ggg:
         ggg.watch(x)
         with tf.GradientTape() as gggg:
            gggg.watch(x)
            u = self.model(x)
         du_dx = gggg.gradient(u, x)
      d2u_dx2 = ggg.gradient(du_dx, x)
   d3u_dx3 = gg.gradient(d2u_dx2, x)
d4u_dx4 = g.gradient(d3u_dx3, x)
return u, d2u_dx2, d4u_dx4
```

In the main, the build function is initially called by the module PINN, in order to define the two networks u_model and PINN_model. The target input and output vectors for the PINN_model network of this problem are defined, according to which the training will take place. Finally, the compile and fit functions where the training will take place are called. The above processes are implemented sequentially through the commands shown in the following lines.

```
u_model, PINN_model = PINN(w, L, E, I).build()

# create training input
x_1 = L*np.random.rand(num_train_samples, 1)        # x_1 = 0 ~ L
x_2 = np.random.rand(num_train_samples, 1)
x_2 = L*np.round(x_2)                               # x_2 = 0 or L
x_3 = np.random.rand(num_train_samples, 1)
x_3 = L*np.round(x_3)                               # x_3 = 0 or L
```

```
# create training output
u_1 = np.zeros((num_train_samples, 1))          # u_1 = 0
u_2 = np.zeros((num_train_samples, 1))          # u_2 = 0
u_3 = np.zeros((num_train_samples, 1))          # u_3 = 0

x_train = [x_1, x_2, x_3]
y_train_target = [u_1, u_2, u_3]\

PINN_model.compile(optimizer='adam',
                   loss=tf.keras.losses.mse,
                   metrics=[tf.keras.metrics.mse],
                   )
model_history = PINN_model.fit(x_train, y_train_target, batch_size=64, epochs=3000, callbacks=callbacks)
```

The structure of the PINN is depicted in Figure **4.5**, and the numerical results are presented in Figures **4.6** and **4.7** and in Table **4.3**.

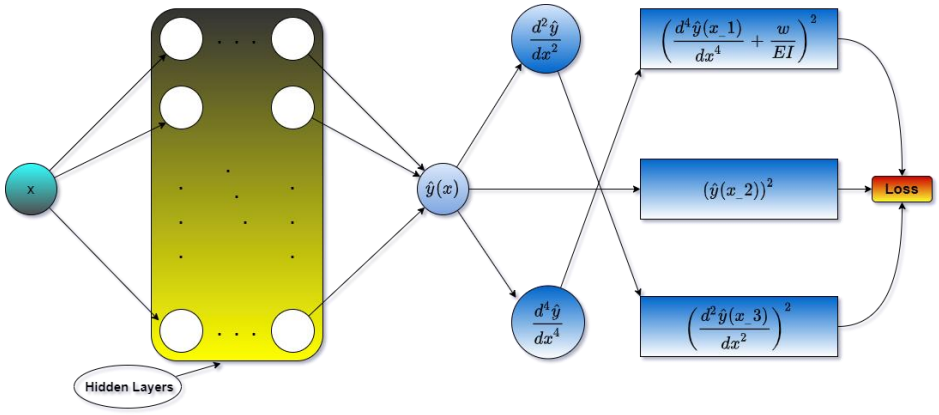


Figure 4.5*:* Illustration of the two networks for Problem 2.

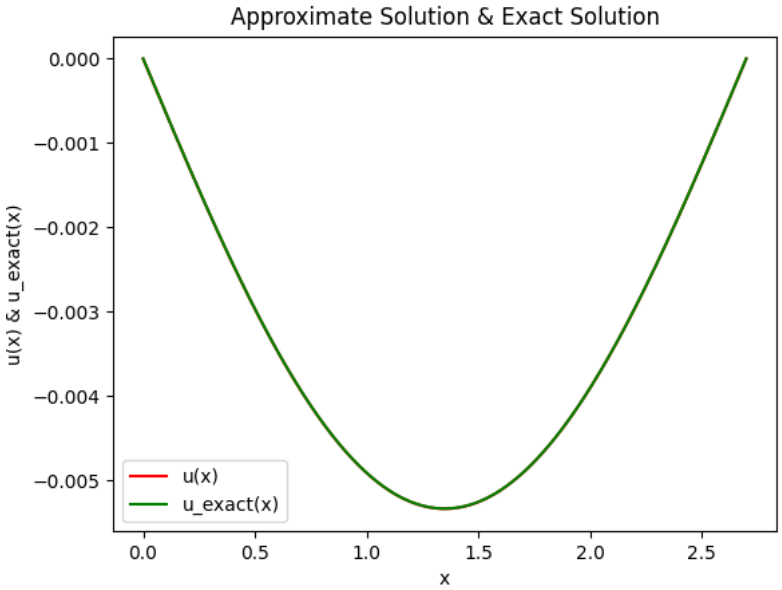


Figure 4.6*:* Graphical representation– No 4 from Table **4.3**.

Table 4.3: The results of solving Problem 2 using Adam's algorithm.

No.	Layers	Batch Size	Epochs	Training Sample	Loss Mse	Time(Sec)
1	[15, 30, 60]	128	3000	10000	1.84E-08	1763
2	[15, 30, 60]	128	870	10000	9.80E-11	491
3	[15, 30, 60, 30]	64	398	10000	9.96E-11	429
4	[15, 30, 60, 30, 15]	64	398	10000	6.74E-11	61
5	[15, 30, 60, 30, 15]	64	476	10000	9.67E-11	839
6	[15, 30, 60, 30, 15]	64	560	10000	9.97E-11	992
7	[15, 30, 15]	64	555	10000	8.52E-11	517
8	[15, 30, 15]	64	1456	10000	6.24E-11	1308
9	[15, 30, 15]	32	395	10000	7.75E-11	713
10	[40, 40]	32	1701	1000	9.25E-11	239

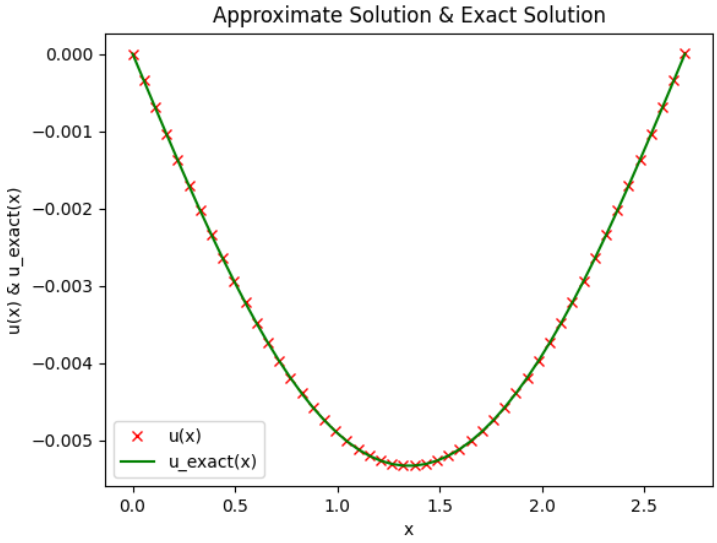


Figure 4.7*:* Indicative graph of the analytical solution and the discrete values of the approximate solution, provided by the ANN, trained with error 9.9340e-11.

Problem 3: Cantilever Beam (Figure **4.8**).

In this example, we consider the differential equation describing the behavior of a cantilever beam in bending.

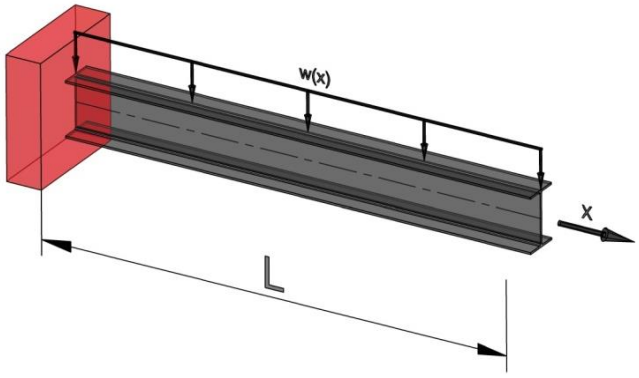


Figure 4.8: Model of a cantilever beam.

The equation is the same as in Problem 2, i.e., (4.12) holds. The boundary conditions for the fixed edge $x = 0$ are

$$y(0) = 0, \quad \frac{dy(0)}{dx} = 0. \tag{4.15}$$

For the free edge $x = L$, it is true $M(L) = 0, V(L) = 0$. Therefore,

$$\frac{d^2y(L)}{dx^2} = 0, \quad \frac{d^3y(L)}{dx^3} = 0. \tag{4.16}$$

The exact solution $y(x)$ of the problem (4.12), (4.15) and (4.16) is given by

$$y(x) = \frac{w}{24EI}(-x^4 + 4Lx^3 - 6\,L^2x^2). \tag{4.17}$$

Below, the required scripts in Python for computing the derivatives in the governing equation (4.12) and the boundary conditions (4.15) and (4.16) are outlined. The scientific software for neural networks Tensorflow in Python, and Numerical computing tools Numpy is used. Here one needs to change the input-output vectors in the main. The quantities u_1, u_2, u_3, u_4, u_5 which correspond to the equation (4.12) and the boundary conditions (4.15), (4.16). These processes are performed through the commands of the following lines of code.

```
# create training input
x_1 = L*np.random.rand(num_train_samples, 1)        # x_1 = 0 ~ L
x_2 = np.zeros((num_train_samples, 1))              # x_2 = 0
x_3 = np.zeros((num_train_samples, 1))              # x_3 = 0
x_4 = L*np.ones((num_train_samples, 1))             # x_4 = L
x_5 = L*np.ones((num_train_samples, 1))             # x_5 = L
# create training output
u_1 = np.zeros((num_train_samples, 1))          # u_1 = 0
u_2 = np.zeros((num_train_samples, 1))          # u_2 = 0
u_3 = np.zeros((num_train_samples, 1))          # u_3 = 0
u_4 = np.zeros((num_train_samples, 1))          # u_4 = 0
u_5 = np.zeros((num_train_samples, 1))          # u_5 = 0

with tf.GradientTape() as g:
  g.watch(x)
  with tf.GradientTape() as gg:
    gg.watch(x)
    with tf.GradientTape() as ggg:
      ggg.watch(x)
      with tf.GradientTape() as gggg:
        gggg.watch(x)
        u = self.model(x)
      du_dx = gggg.gradient(u, x)
    d2u_dx2 = ggg.gradient(du_dx, x)
  d3u_dx3 = gg.gradient(d2u_dx2, x)
d4u_dx4 = g.gradient(d3u_dx3, x)
return u, du_dx, d2u_dx2, d3u_dx3, d4u_dx4

# compute gradients
u, du_dx, d2u_dx2, d3u_dx3, d4u_dx4 = grads(x_1)
# equation output being zero
```

```
u_1 = d4u_dx4 + self.w/(self.E * self.I)
# initial condition output
u_2 = u_model(x_2)
u, du_dx, d2u_dx2, d3u_dx3, d4u_dx4 = grads(x_3)
u_3 = du_dx
u, du_dx, d2u_dx2, d3u_dx3, d4u_dx4 = grads(x_4)
u_4 = d2u_dx2
u_5 = d3u_dx3
```

The architecture of the PINN is illustrated in Figure **4.9**, and the numerical results are presented in Figures **4.10** and **4.11** and in Table **4.4**.

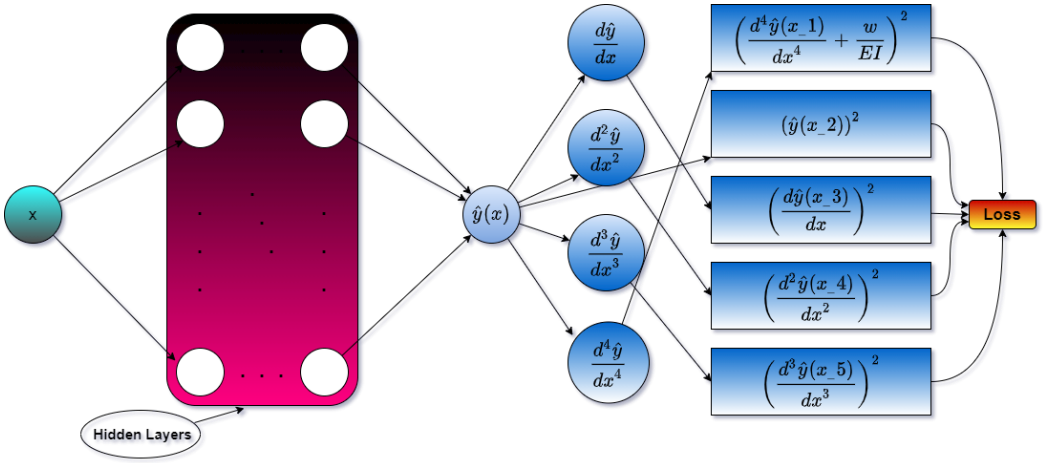


Figure 4.9: Illustration of the two networks for problem 3.

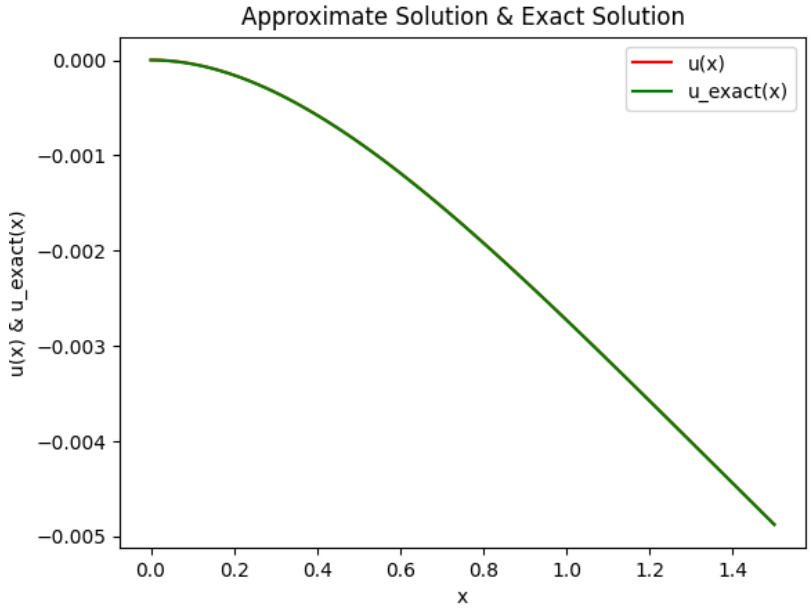


Figure 4.10*:* Graphical representation – No 7 from Table 4.4.

Table 4.4: The results of solving Problem 3 with using Adam's algorithm.

No.	Layers	Batch Size	Epochs	Training Sample	Loss Mse	Time(Sec)
1	[15, 30, 60]	64	1000	1000	8.45E-10	360
2	[15, 30, 60]	32	1000	10000	6.66E-11	2000
3	[15, 30, 60]	32	1000	1000	2.47E-07	500
4	[15, 30, 60]	32	3000	1000	6.70E-10	720
5	[15, 30, 60]	32	4000	1000	6.24E-09	1020
6	[15, 30, 60]	32	1000	5000	4.15E-08	1080
7	[15, 30, 60]	32	1000	10000	1.60E-11	1860
8	[15, 30, 60]	32	900	10000	3.16E-08	1740
9	[15, 30, 60]	32	1000	10000	1.58E-10	1500
10	[15, 30, 60]	32	702	10000	7.44E-11	1200

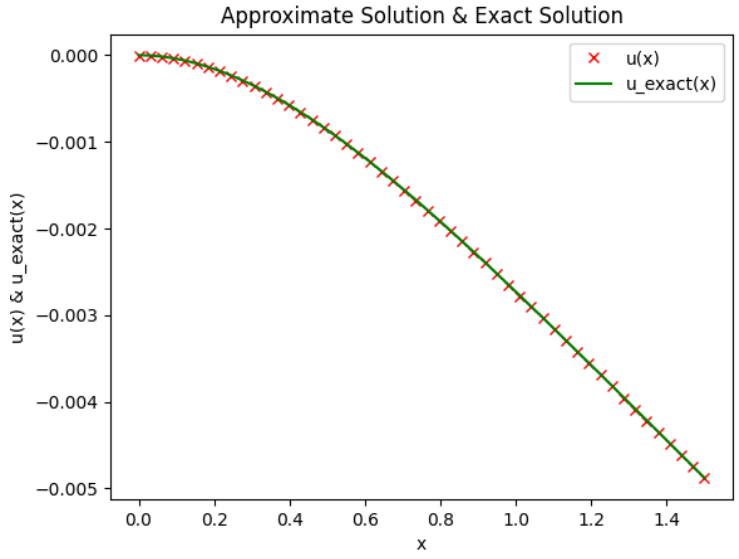


Figure 4.11: Indicative graph of the analytical solution and the discrete values of the approximate solution, provided by the ANN, trained with error 8.7555e-11.

Problem 4: Cantilever, Simply Supported Beam (Figure **4.12**)

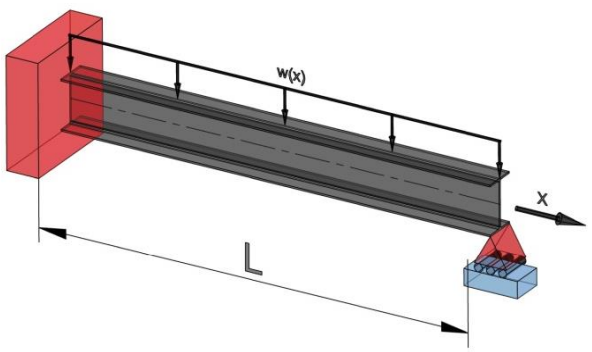


Figure 4.12*:* Model of a beam with anchoring at the left end ($x = 0$) and simply supported at the right end ($x = L$).

In this problem, the boundary conditions associated with one fixed edge and the supported the other one of the beam are as follows. At the left end, it holds that $y(0) = 0$ and $dy(0)/dx = 0$ while at the right end, it holds that $y(L) = 0$ and $M(L) = 0$. The differential equation is the same as the equation in Problems 2 and 3, i.e. (4.12). The boundary conditions of the present problem are

$$y(0) = 0, \quad \frac{dy(0)}{dx} = 0, \quad y(L) = 0, \quad \frac{d^2y(L)}{dx^2} = 0 \tag{4.18}$$

The exact solution of the differential equation (4.12) with the boundary conditions (4.18) is

$$y(x) = \frac{w}{24\,EI}\left(-x^4 + \frac{5}{2}\,Lx^3 - \frac{3}{2}x^2\right). \tag{4.19}$$

As in Problem 3, some modifications in the Python code that implements the PINN method are needed. For this problem, it is necessary to create 5 input vectors x_1, x_2, x_3, x_4, x_5 which are needed for training (num_train_samples). At the same time, the corresponding output vectors must be created - target u_1, u_2, u_3, u_4, u_5 which, as shown by (4.12), (4.18) must consist of zero elements. The corresponding scripts are listed below.

```
# create training input
x_1 = L*np.random.rand(num_train_samples, 1)    # x_1 = 0 ~ L
x_2 = np.zeros((num_train_samples, 1))          # x_2 = 0
x_3 = np.zeros((num_train_samples, 1))          # x_3 = 0
x_4 = L*np.ones((num_train_samples, 1))         # x_4 = L
x_5 = L*np.ones((num_train_samples, 1))         # x_5 = L
# create training output
u_1 = np.zeros((num_train_samples, 1))          # u_1 = 0
u_2 = np.zeros((num_train_samples, 1))          # u_2 = 0
u_3 = np.zeros((num_train_samples, 1))          # u_3 = 0
u_4 = np.zeros((num_train_samples, 1))          # u_4 = 0
u_5 = np.zeros((num_train_samples, 1))          # u_5 = 0

return u, du_dx, d2u_dx2, d4u_dx4

grads = GradientLayer(u_model)

# compute gradients
u, du_dx, d2u_dx2, d4u_dx4 = grads(x_1)
# equation output being zero
u_1 = d4u_dx4 + self.w/(self.E * self.I)
# initial condition output
u_2 = u_model(x_2)
u, du_dx, d2u_dx2, d4u_dx4 = grads(x_3)
u_3 = du_dx
u, du_dx, d2u_dx2, d4u_dx4 = grads(x_4)
u_4 = u_model(x_4)
u_5 = d2u_dx2
```

The architecture of the PINN is illustrated in Figure **4.13**, and the numerical results are presented in Figures **4.14** and **4.15** and Table **4.5**.

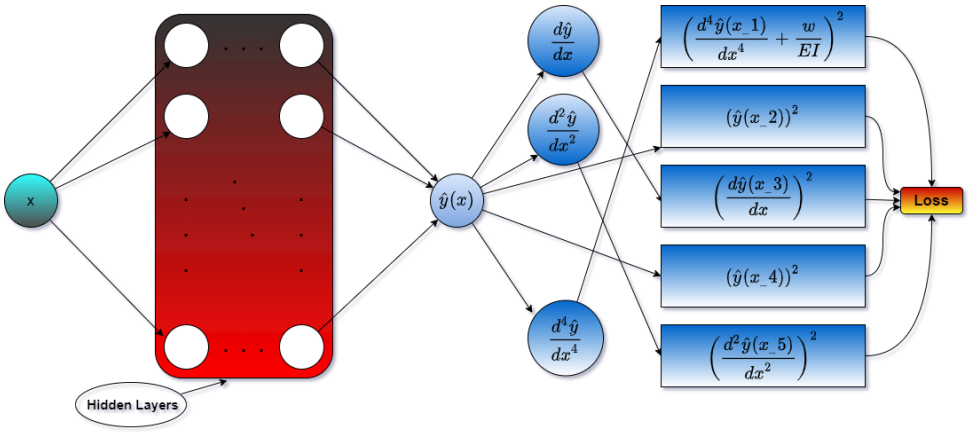


Figure 4.13*:* Illustration of the two networks for Problem 4.

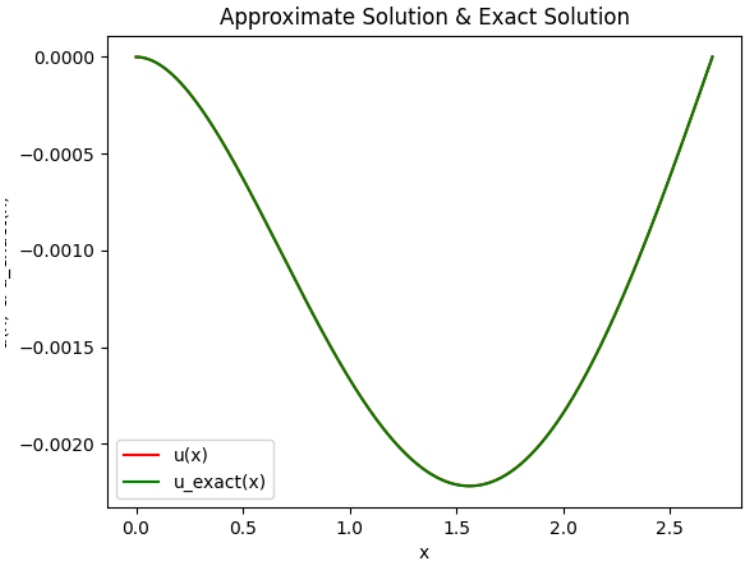


Figure 4.14*:* Graphical representation – No 1 from Table 4.5.

Table 4.5: The results of solving Problem 4 using Adam's algorithm.

No.	Layers	Batch Size	Epochs	Training Sample	Loss Mse	Time(Sec)
1	[15, 30, 60]	32	676	10000	7.70E-11	1504
2	[15, 30, 60]	32	3000	1000	1.93E-08	695
3	[15, 30, 60]	32	2446	1000	9.82E-11	608
4	[15, 30, 60]	16	2519	1000	9.99E-11	1036
5	[15, 30, 60]	64	3000	1000	1.11E-08	414
6	[15, 30, 60]	64	789	10000	7.98E-11	1002
7	[25, 50, 25]	64	900	10000	9.09E-11	1054
8	[25, 50, 25]	128	1021	10000	9.96E-11	1555
9	[25, 50, 25]	128	3000	1000	1.04E-07	215
10	[25, 50, 25]	64	3000	1000	6.84E-10	302
11	[25, 50, 50, 25]	16	2083	1000	9.26E-11	1053

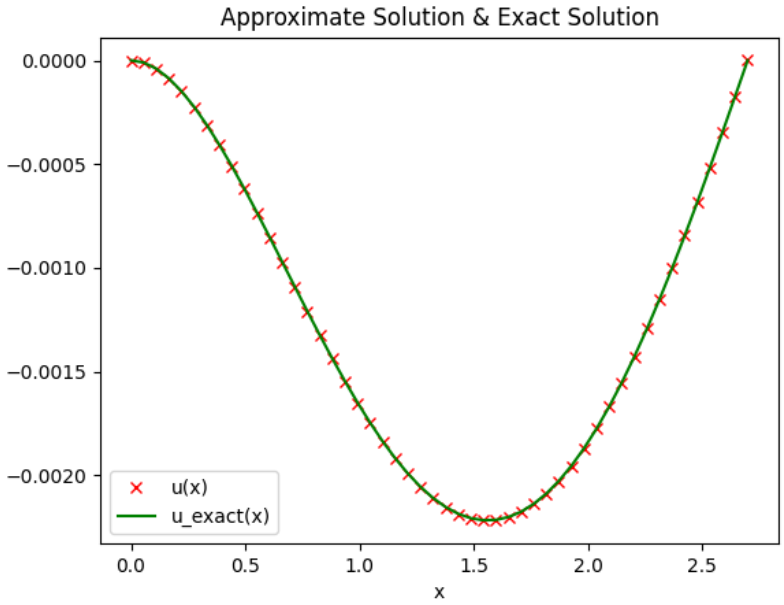


Figure 4.15: Indicative graph of the analytical solution and the discrete values of the approximate solution, provided by the ANN, trained with error 7.4316e-11.

4.2. Discussions

Problem 1 was solved by the L-BFGS algorithm and with by Adam algorithm. Problems 2, 3, and 4 were solved only with the program using the Adam algorithm. In all implementations, the hyperbolic tangent tanh(x) was used as an activation function. The choice of numerical accuracies has been based on trial and error and previous information from the literature.

5. Remarks and Conclusions

The mechanical boundary value problems for a rod and a beam have been solved numerically with the application of PINN. The computational techniques have been described in detail, and the main programming parts of the code, based on the Tensorflow scientific software in Python for neural networks for each investigated model, have been presented. The results obtained after solving the considered problems are promising. As the numerical results have shown, the usage of Adam's optimization method within the Keras library seems to be more effective than the L-BFGS algorithm. The rate of convergence and the accuracy of the approximate solution can be improved by increasing the number of epochs, the batch size, the number of collocation points etc.

Even though the problems considered here are academic examples, i.e., the exact solutions exist, the proposed techniques can be easily modified and applied to more complex mechanics problems with unknown analytical solutions. The introduced techniques, together with programming snippets, can be very helpful for readers in order to understand the steps of implementation of PINN for mechanical models. They also can be easily extended to dynamic problems, direct and inverse (parameter identification) problems, and contact problems in continuum mechanics.

Nomenclature

PINN	=	Physics-informed neural network
$w_{k,ij}$	=	Weight from i - neuron to j – neuron of layer k
b_{kj}	=	Biases to j – neuron of layer *k*
N_L	=	Number of layers

n_k	=	Number of neurons in layer k
$NN\ (x)$	=	Output of the neural network
$\hat{u}(x)$	=	Approximate solution
$u(x)$	=	Exact solution
L	=	Total length of the rod/beam
C	=	Constant coefficient
A	=	Cross sectional area
ε	=	Strain in the rod/beam
σ	=	Stress in the rod/beam
E	=	Young's modulus
$M(x)$	=	Bending moment
P	=	Concentrated load
I	=	Moment of inertia
L-BFGS	=	Limited- Broyden–Fletcher–Goldfarb–Shann
Adam	=	Adaptive moment estimation

References

[1] Avdelas AV, Panagiotopoulos PD, Kortesis S. Neural networks for computing in the elastoplastic analysis of structures. Meccanica 1995; 30: 1–15.

[2] Baydin A, Pearlmutter BA, Radul AA, Siskind JM. Automatic Differentiation in Machine Learning: a Survey, 2018, https://arxiv.org/pdf/1502.05767.pdf

[3] Cande's EJ. Harmonic Analysis of Neural Networks, Applied and Computational Harmonic Analysis 1999; 6: 197–218.

[4] Fletcher R. Practical Methods of Optimization (2nd ed.). John Wiley & Sons, New York 1987.

[5] Guo M, Haghighat E. An energy-based error bound of physics-informed neural network solutions in elasticity. arXiv preprint arXiv:2010.09088, 2020.

[6] Kadeethum T, Jørgensen T, Nick H. Physics-informed neural networks for solving nonlinear diffusivity and Biot's equations. PLoS ONE 2020, 15(5): e0232683.

[7] Karniadakis GE, Kevrekidis IG, Lu L, *et al*. Physics-informed machine learning. Nat Rev Phys 2021; 3: 422–440. https://doi.org/10.1038/s42254-021-00314-5.

[8] Kingma DP, Ba JL. Adam A. A Method for Stochastic Optimization, Computer Science, Mathematics, The International Conference on Learning Representations (ICLR) 2015.

[9] Kovachki N, Lanthaler S, Mishra S. On Universal Approximation and Error Bounds for Fourier Neural Operator, Journal of Machine Learning Research 2021; 22: 1-76.

[10] Kortesis S. Panagiotopoulos PD. Neural networks for computing in structural analysis: Methods and prospects of applications. International Journal for Numerical Methods in Engineering 1993; 36: 2305-2318.

[11] Lagaris E, Likas A, Fotiadis DI. Artificial neural networks for solving ordinary and partial differential equations, IEEE Transactions on Neural Networks 1998; 9: 987–1000.

[12] Lee H, Kang I. Neural algorithms for solving differential equations. Journal of Computational Physics 1990; 91(1): 110-131.

[13] Li Z, Kovachki N, Azizzadenesheli K, Liu B, Bhattacharya K, Stuart A, Anandkumar A. Fourier neural operator for parametric partial differential equations, 2020, arXiv preprint arXiv:2010.08895.

[14] Lu L, Jin P, Karniadakis GE. DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators, 2020, https://arxiv.org/abs/1910.03193

[15] Lu L, Jin P, Pang G, Zhang Z, Karniadakis GE. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. Nat Mach Intell 2021; 3: 218–229. https://doi.org/10.1038/s42256-021-00302-5.

[16] McCulloch WS, Pitts W. A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 1990; 52(1/2): 99-115.

[17] Meade Jr AJ, Fernadez AA. The numerical solution of linear ordinary differential equations by feedforward neural networks. Math. Comput. Modelling. 1994; 19(12): 1-25.

[18] Muradova AD, Stavroulakis GE. The projective-iterative method and neural network estimation for buckling of elastic plates in nonlinear theory. Communications in Nonlinear Science and Numerical Simulation 2007; 12: 1068-1088.

[19] Muradova AD, Stavroulakis GE. Physics-informed neural networks for elastic plate problems with bending and Winkler-type contact effects. Journal of the Serbian Society for Computational Mechanics 2021 ; 15(2): 45-54.

[20] Peng W, Zhang J, Zhou W, Zhao X, Yao W, Chen X. A Physics-Informed Neural Network Library 2021: https://arxiv.org/pdf/2107.04320.pdf.

[21] Raissi M, Perdikaris P, Karniadakis GE. Physics Informed Deep Learning (Part I): Data - driven Solutions of Nonlinear Partial Differential Equations 2017: https://arxiv.org/abs/1711.10561.

[22] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics 2019; 378: 686–707.

[23] Rojas R. Neural Networks A Systematic Introduction. Springer-Verlag, Berlin, 1996.

[24] Ruder S. An overview of gradient descent optimization algorithms 2017: https://arxiv.org/abs/1609.04747.

[25] Russel S, Norvig P. Artificial Intelligence: A modern approach (Pearson Series in Artifical Intelligence), Pearson, 4th Edition, 2020.

[26] Searle JR. The Rediscovery of the Mind. Cambridge, Massachusetts: MIT Press, 1992.

[27] Shin Y, Darbon J, Karniadakis GE. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. Commun. Comput. Phys. 2020; 28: 2042–2074.

[28] Stavroulakis GE. Inverse and Crack Identification Problems in Engineering Mechanics. Springer, 2000.

[29] Stavroulakis GE, Avdelas A, Abdalla KM, Panagiotopoulos PD. A neural network approach to the modelling, calculation and identification of semi-rigid connections in steel structures, Journal of Constructional Steel Research 1997; 44(1–2): 91-105.

[30] Tartakovsky AM, Marrero CO, Perdikaris P, Tartakovsky GD, Barajas-Solano D. Learning parameters and constitutive relationships with physics informed deep neural networks 2018: arXiv preprint arXiv:1808.03398.

[31] Waszczyszyn Z, Ziemiański L. Neural Networks in the Identification Analysis of Structural Mechanics Problems. In: Mróz Z, Stavroulakis G.E. (eds) Parameter Identification of Materials and Structures, CISM International Centre for Mechanical Sciences (Courses and Lectures), 2005; 469, Springer, Vienna.

[32] Yagawa G, Oishi A. Computational mechanics with neural networks. Springer, 2021.

[33] Zhang Q, Chen Y, Yang Z. Data-driven solutions and discoveries in mechanics using physics informed neural network. 2020.

[34] Zhao X, Gong Z, Zhang Y, Yao W, Chen X. Physics-informed Convolutional Neural Networks for Temperature Field Prediction of Heat Source Layout without Labeled Data 2021, https://arxiv.org/abs/2109.12482.