UM6P University Mohammed VI Polytechnic

College of Computing

1st year master thesis

Physics Informed Neural Networks for beam analysis

by

Othmane OUKBIL

Supervisor and first examiner :

Pr. Khalil FERRADI

Department:

College of computing
Modeling option

July 19, 2024

Summary and keywords

Deep learning has achieved remarkable success in diverse applications; however, its use in solving partial differential equations (PDEs) has emerged only recently. Here, we present an overview of physics-informed neural networks (PINNs), which embed a PDE into the loss of the neural network using automatic differentiation. The PINN algorithm is simple, and it can be applied to different types of PDEs, including integro-differential equations, fractional PDEs, and stochastic PDEs. Moreover, from an implementation point of view, PINNs solve inverse problems as easily as forward problems. We also present a Python library for PINNs, DeepXDE, which is designed to serve both as an educational tool to be used in the classroom as well as a research tool for solving problems in computational science and engineering. Specifically, DeepXDE can solve forward problems given initial and boundary conditions, as well as inverse problems given some extra measurements. DeepXDE supports complex-geometry domains based on the technique of constructive solid geometry and enables the user code to be compact, resembling closely the mathematical formulation. We employed the DeepXDE library to implement Physics-Informed Neural Networks (PINNs) in the context of three distinct nonlinear beam equations.

keywords

Machine learning, Neural Networks, Physics-Informed Neural Networks(PINNs), Euler-Bernoulli beam theory, partial differential equations, DeepXDE.

Abstract

Physics-Informed Neural Networks (PINN) are neural networks (NNs) that encode model equations, like Partial Differential Equations (PDE), as a component of the neural network itself. PINNs are nowadays used to solve PDEs, fractional equations, integral-differential equations, and stochastic PDEs. This novel methodology has arisen as a multi-task learning framework in which a NN must fit observed data while reducing a PDE residual. This report investigates the application of PINNs to solve nonlinear beam equations, which are fundamental in engineering for understanding beam behavior under various constraints. Initially, an analytical exploration is conducted to understand the capabilities of machine learning techniques in solving PDEs. The methodology of PINNs, which integrates physical laws directly into the neural network framework, is explained in detail. The study then applies the PINNs method to the Euler-Bernoulli static equations for beams using the DeepXDE library. This includes analyzing three types of beams: cantilever beams, simply supported beams, and clamped beams. The results highlight the effectiveness of PINNs in accurately solving these beam equations, demonstrating a promising approach for tackling complex PDEs in various engineering applications where traditional numerical methods might struggle, particularly in scenarios with incomplete or uncertain data.

Introduction

Machine learning has been around for decades. No one knows for sure the first time it was used, but its history is often recounted with several key events. A 1943 paper by logician and cognitive psychologist Walter Pitts and American neurophysiologist Warren McCulloch attempted to mathematically map out thought processes and decision-making in human cognition. The two provided a way to describe brain functions in abstract terms and showed that simple elements connected in a neural network can have immense computational power.

Seven years later, in 1950, English mathematician Alan Turing developed his "Turing Test," meant to determine if a computer has real intelligence. According to Turing, the question whether machines can think is itself "too meaningless" to deserve discussion. Instead, he reasoned, it was worthwhile to determine whether a digital computer can do well in a certain kind of game that Turing describes ("The Imitation Game"). In this case, a remote human interrogator within a fixed time frame must distinguish between a computer and a human subject based on their replies to various questions. According to Turing, a computer's success at "thinking" can be measured by its probability of being misidentified as the human subject.

Two years later, Arthur Samuel wrote the first computer learning program, the game of checkers. The IBM computer used in this experiment improved its performance the more it played, showing it had, in fact, learned. Later that same decade, in 1957, Frank Rosenblatt designed the first neural network for computers—the perceptron—which simulated the thought processes of the human brain.

In 1981, Gerald Dejong introduced the concept of "Explanation Based Learning," in which a computer analyzed training data and created a general rule it could follow by discarding unimportant information. A decade later, work in this area underwent an important transformation, shifting from a knowledge-driven approach to a data-driven approach. It was then that scientists started creating programs for computers to analyze large amounts of data and draw conclusions from the results.

In 2006, Geoffrey Hinton coined the term “deep learning” to explain new algorithms that let computers “see” and distinguish objects and text in images and videos. Five years later, IBM’s Watson beat a human competitor on the TV trivia show Jeopardy!.

The year 2011 saw the invention of Google Brain. Its deep neural network was able to discover and categorize objects, and shortly after it was developed, the system had taught itself to recognize cats.

Machine learning is key to the development of self-driving automobiles. Although these vehicles are not yet part of our everyday lives—there are far fewer of them on the road than was predicted even five years ago—they are still a subject of great fascination and a focus of major car manufacturers.

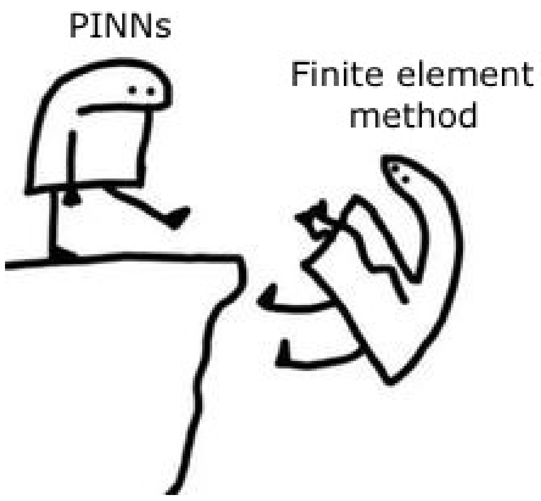
Despite this delay in the automotive industry, machine learning is already inside our homes. It’s what is at work behind online recommendations—think the products pushed on Amazon and shows promoted by Netflix—and is critical in fraud detection.

Physics-informed machine learning debuted in the 1990s, appearing in scattered papers throughout the decade. A resurgence in machine learning around 2010 breathed new life into this promising offshoot.

Another milestone came in 2014 with Facebook’s DeepFace. Its algorithm was able to detect whether two faces in unfamiliar photos are of the same person with 97.25% accuracy, despite differing lighting conditions or angles. Humans generally have an average of 97.53% accuracy, meaning Facebook’s facial-processing software had nearly the same accuracy as a human being.

List of Figures

Contents



Could this happen?

Chapter 1

Fundamental Concepts of Machine Learning

1.1 GENERAL Q&A

How can we collect data?

By observing it in nature, handcrafting it by humans, or generating it by another algorithm.

What is a computer program?

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

What are the features of a machine learning algorithm?

A dataset, a cost function, an optimization procedure, and a parameterized model.

What are the types of learning?

Supervised, Unsupervised, Semi-supervised, and Reinforcement Learning.

What is Supervised Learning?

The algorithm processes a labeled dataset.

What is Unsupervised Learning?

The data is not labeled and therefore no explicit prediction is possible.

What is Semi-supervised Learning?

Only small samples of the data are labeled.

What is Reinforcement Learning?

It is used for problems involving sequential decision-making in order to fulfill a long-term goal.

1.2 Machine Learning Tasks

Regression

Regression is a supervised learning problem with the goal of predicting a numerical value. For example, linear regression:

$$y = \mathbf{w}^T \mathbf{x} + b$$

Classification

Classification tasks belong to the category of supervised learning. Instead of a numerical value, their output takes on a discrete form. For example, K-Nearest Neighbors (KNN). In other words, a classification algorithm returns a function that assigns a category to the provided input.

Clustering

Clustering differs from classification and regression as it is an unsupervised learning task. The algorithm gives feedback about which parts of the data share similarities and therefore belong to the same cluster. A popular choice for clustering is the k- means algorithm that divides the incoming data into k different clusters of examples being close to each other.

1.3 Optimization Techniques in Machine Learning

Generally, the goal of an optimization is to minimize or maximize an optimization criterion or objective function $C(\Theta)$ by altering Θ. In our context, the function is the cost function, and Θ denotes the parameters of a model, normally the weights w and biases b.

If the optimization criterion is differentiable, we can use gradient descent.

For neural networks, the optimization problem is non-convex and hence it is more likely to converge to a local minimum.

It is difficult to find a global minimum in non-convex optimization problems.

A major drawback of gradient descent is its sensitivity to the choice of the learning rate α. If α is too large, the algorithm starts to oscillate or even fails to converge at all. Conversely, a small α value leads to an extremely low convergence rate.

In practice, stochastic gradient descent often shows better convergence properties.

The most popular optimization approach in practice is mini-batch gradient descent, which uses a compromise between full-batch and stochastic gradient descent. The idea is to find an approximation of the gradient by evaluating the gradient for a small sample of the data.

1.4 Overfitting Versus Underfitting

We are in an overfitting situation when the capacity of a model can be too high, meaning the algorithm chooses a complex function that fits the training data perfectly but fails to generalize because it overestimates the importance of noisy data. Secondly, a model with a very low capacity is applied to a highly non-linear problem. For example, if the hypothesis space of a model only contains linear functions, it is not able to represent data that is following quadratic or cubic functions. This case describes underfitting.

A way to tackle the problem of underfitting is to increase the set of functions an algorithm is allowed to select.

1.5 Regularization

The previous section showed a possibility to deal with underfitting by increasing the number of features x_i. In the same way, overfitting can be reduced when features are removed from the model. However, this option is not very popular, since removing features means that information about the problem gets lost. Additionally, adding more examples to the training set can help in overfitting, but in most cases, it's impossible to gather more data about the problem.

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error (low variance) but not its training error (low bias).

Let's take the cost function for linear regression as an example:

Lasso Regularization

$$\hat{C}(w,b) = C(w,b) + \lambda ||w||_1$$

Adding a term proportional to the magnitude of the weights forces the algorithm to select a model with smaller weights. The influence of the penalty term is controlled by λ.

Ridge Regularization

$$\hat{C}(w,b) = C(w,b) + \lambda w^T w$$

An advantage of $L2$ (ridge) regularization is that the penalty term is differentiable.

Apart from model selection and adding a penalty term to the objective function, there exist many other techniques to implicitly and explicitly express preferences for a certain solution. A few more are explained in Chap. 2 about neural networks

Chapter 2

Neural Networks

Inspired by the neurons of the brain, an early form, known as perceptron, was created by Frank Rosenblatt in 1958. With the introduction of the backpropagation algorithm in 1986, the learning capabilities of neural networks improved significantly. A neural network is like any other supervised learning model.

2.1 Feed-Forward Neural Network

A neural network is like any other supervised learning model, just a parameterized function defining a mapping $y = f_n(x) = f_3(f_2(f_1(x)))$, where each function f_i represents a layer of the network.

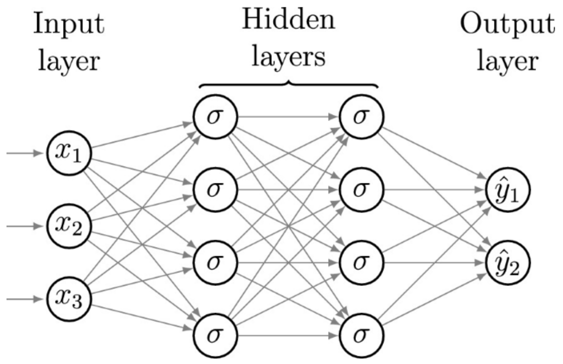


Figure 2.1: a fully connected Feed-Forward Neural Network

The information in the form of input x flows from the input layer through an arbitrary number of so-called "hidden" layers to the output layer, hence the name feed-forward neural network. The circles represent the basic units called "neurons." A single neuron receives a vector of features x and produces a scalar output $a(x)$, which serves as an input for the neurons in the next layer.

The output of a neuron can be decomposed into two operations. First, the input vector x is transformed into:

1. $z = w^T x + b$
2. $a(x) = \sigma(z)$

The non-linear function σ is called the activation function and is usually chosen to be the same for all neurons. Depending on the function chosen for the output neurons, neural networks are able to solve either regression or classification tasks. Generally, a fully connected feed-forward neural network with one hidden layer is capable of approximating any continuous multi-input/multi-output function with arbitrary precision, given that the hidden layer contains a sufficient amount of neurons.

2.2 Forward Propagation

It is important to get a better understanding of how the data is propagated from the input to the output layers. The output for the current neuron j in the l-th layer is calculated by the following formula:

$$a_j^{(l)} = \sigma(z_j^{(l)}) = \sigma\left(\sum_k (w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)})\right)$$

Using only matrix and vector notation:

$$a^{(l)} = \sigma(z^{(l)}) = \sigma(W^{(l)} a^{(l-1)} + b^{(l)})$$

2.3 Differentiation

Up to this point, the neural network was only able to transform an input into an output depending on the randomly initialized set of weights and biases. This means the network has not utilized any data to learn the optimal parameters w and b that generate the desired output.

The mean squared error loss is chosen to quantify the prediction accuracy of the model. The cost function takes on the familiar form:

$$C = \frac{1}{2m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

The factor $\frac{1}{2}$ is to simplify the expression of the derivative needed for later calculations.

To find optimal network parameters, the cost function is minimized with the gradient descent method. In engineering applications, a common approach for the numerical approximation of derivatives is the method of finite differences. However, this approach introduces truncation errors and quickly becomes computationally expensive for large networks with many parameters.

Automatic differentiation is a more efficient alternative allowing the fast and precise numerical evaluation of higher-order derivatives.

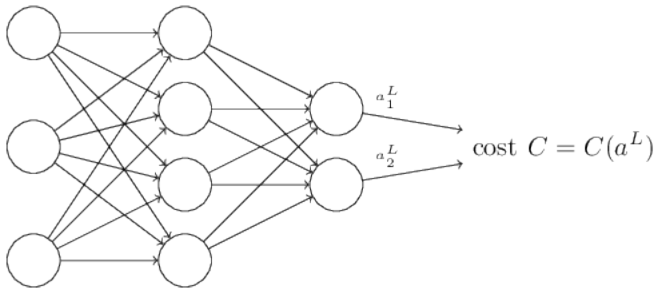
2.4 Backpropagation

The goal of backpropagation is to compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to any weight w or bias b in the network. It belongs to the field of automatic differentiation. The expression tells us how quickly the cost changes when we change the weights and biases. For backpropagation to work we need to make two main assumptions about the form of the cost function. The first assumption we need is that the cost function can be written as an average:

$$C = \frac{1}{m} \sum_{x} C_x$$

over cost functions C_x for individual training examples, m. The reason we need this assumption is because what backpropagation actually lets us do is compute the partial derivatives $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example. We then recover $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging over training examples.

The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network.



For example, the quadratic cost function satisfies this requirement, since the quadratic cost for a single training example x may be written as

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2,$$

and thus is a function of the output activations. Of course, this cost function also depends on the desired output y, and you may wonder why we're not regarding the cost also as a function of y. Remember, though, that the input training example x is fixed, and so the output y is also a fixed parameter. In particular, it's not something we can modify by changing the weights and biases in any way, i.e., it's not something which the neural network learns. And so it makes sense to regard C as a function of the output activations a^L alone, with y merely a parameter that helps define that function.

2.5 Activation function

Activation functions play an integral role in neural networks by introducing nonlinearity. This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear regression model.

Many different nonlinear activation functions have been proposed throughout the history of neural networks. In this post, you will explore three popular ones: sigmoid, tanh, and ReLU. why do we need non-linear activation functions? the answer is to make the network represent more complex functions.

Sigmoid function

Let's start with a popular example, the sigmoid function. It is a popular choice for the nonlinear activation function for neural networks. One reason it's popular is that it has output values between 0 and 1, which mimic probability values. Hence it is used to convert the real-valued output of a linear layer to a probability, which can be used as a probability output. This also makes it an important part of logistic regression methods, which can be used directly for binary classification. Here, we can observe that the

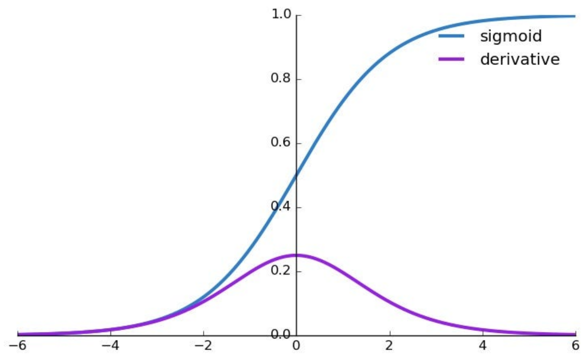


Figure 2.2: Sigmoid activation function (blue) and gradient (purple)

gradient of the sigmoid function is always between 0 and 0.25. And as the x tends to positive or negative infinity, the gradient tends to zero. This could contribute to the vanishing gradient problem, meaning when the inputs are at some large magnitude of x (e.g., due to the output from earlier layers), the gradient is too small to initiate the correction.

The hyperbolic tangent function

The hyperbolic tangent activation function, also known as "tanh" is a widely used activation function in artificial neural networks. It is a non-linear function that takes a real-valued number as input and maps it to a value between -1 and 1. The shape of the tanh function is similar to the sigmoid function, but the range is from -1 to 1 instead of 0 to 1. The tanh function is symmetric around the origin, which means that for negative inputs, it produces negative outputs, and for positive inputs, it produces positive outputs. The output is 0 when the input is 0.

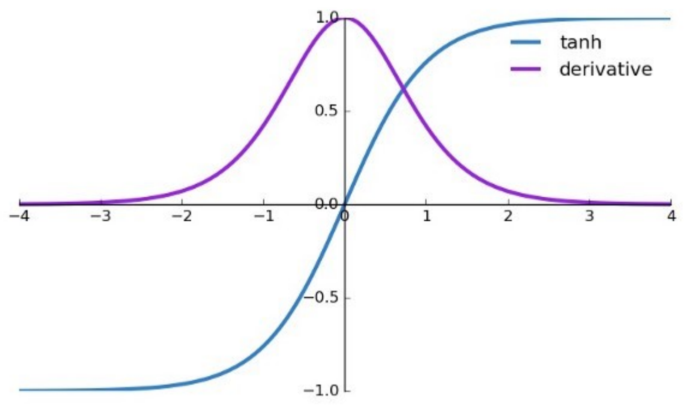


Figure 2.3: Tanh activation function (blue) and gradient (purple)

Notice that the gradient now has a maximum value of 1, compared to the sigmoid function, where the largest gradient value is 0. This makes a network with tanh activation less susceptible to the vanishing gradient problem.

The ReLU function

The ReLU (Rectified Linear Unit) activation function is a commonly used activation function in deep learning, particularly in neural networks. It is a simple yet effective activation function that is computationally efficient and has been shown to perform well in many applications. ReLU generally less effective for PINNs because its second-order derivative is zero, which can limit the model's ability to approximate the PDE solution accurately.

2.6 Regularization of Neural Networks

Regularization is a set of methods for reducing overfitting in machine learning models. Typically, regularization trades a marginal decrease in training accuracy for an increase in generalizability. A definition by Goodfellow states: "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error (low variance) but not its training error (low bias) ." Accordingly, regularization is also known as the bias-variance trade-off.
There are many forms of regularization.

Early Stopping

The principal idea of early stopping is to halt the training process as soon as the validation error rises and the model enters the overfitting regime. Every time the validation error decreases, the weights and biases of the model are stored. The algorithm terminates when the validation error has not improved over a predefined number of iteration steps. Finally, the model parameters at the point of the lowest validation error are returned. Due to its simplicity and effectiveness, early stopping is a very popular regularization method often applied in practice.

L1 and L2 Regularization:

We can try to limit the capacity of models by adding a penalty term $\Omega(\theta)$ to the cost function.

$$\tilde{C}(\theta, X, y) = C(\theta, X, y) + \alpha\Omega(\theta)$$

The α term corresponds to the relative contribution of the regularization penalty to the overall cost/loss .
L2 Regularization :
One of the simplest regularization techniques is weight decay, where $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$. This is also known as ridge regression or Tikhonov regularization.

$$\tilde{C}(\theta, X, y) = C(\theta, X, y) + \frac{\alpha}{2}w^T w$$

Taking the gradient of this cost function yields

$$\nabla_w \tilde{C}(\theta, X, y) = \alpha w + \nabla_w C(\theta, X, y)$$

So for a single gradient step,

$$w \leftarrow (1 - \epsilon\alpha)w - \epsilon\nabla_w C(w; X, y)$$

L1 Regularization :
L1 regularization places a penalty on the absolute values of the weights rather than their squared norm as L2-regularization does. It is defined as

$$\Omega(w) = \sum_i |w_i|.$$

$$\tilde{C}(\theta, X, y) = C(\theta, X, y) + \sum_i |w_i|$$

Taking the gradient of this cost function yields

$$\nabla_w \tilde{C}(\theta, X, y) = \nabla_w C(\theta, X, y) + \lambda \operatorname{sign}(w),$$

where $\operatorname{sign}(w)$ is applied element-wise.

The learning rule for the weights takes on the updated form

$$w \rightarrow w' = w - \alpha\lambda \operatorname{sign}(w) - \alpha \nabla_w \tilde{C}(\theta, X, y)$$

The given formulas only consider the weights, because regularizing the biases can have unwanted effects like leading to severe underfitting.

Theoretically, the hyper-parameter λ could be chosen individually for each layer, but usually it is set equally for the whole network to reduce the search space during hyperparameter tuning.

L1 regularization encourages the selection of few high-importance connections, while the other weights are forced toward zero. By contrast, L2 regularization shrinks the weights proportional to w, so for small weights, the reduction is much smaller compared to L1 regularization and, thus, is rather seen as weight decay.

Dropout

Dropout is a radically different technique for regularization. Unlike L1 and L2 regularization, dropout doesn't rely on modifying the cost function. Instead, in dropout we modify the network itself. With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. We forward- propagate the input x through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network. By repeating this process over and over, our network will learn a set of weights and biases.

Dataset Augmentation

The best way to increase the generalization abilities of a model is to train it on more data. In most cases, obtaining more training data is not feasible, but sometimes the creation of "fake" data can be an option. Another data augmentation technique is the inclusion of noise.

2.7 Input derivatives

Neural networks are fully differentiable functions, it is not only possible to compute the derivatives with respect to the network parameters, but also with respect to the input variables. Even higher- order derivatives can be evaluated and used to check if the solution satisfies the governing partial differential equation of the problem.

Chapter 3

Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks (PINNs) are a class of neural networks that integrate physical laws described by partial differential equations (PDEs) into the learning process. This approach leverages the power of deep learning while ensuring that the solutions adhere to known physical principles. The fusion of neural networks with physics has led to significant advancements in solving complex scientific and engineering problems.
The concept of integrating physical laws with machine learning has been an area of interest for several decades. However, the formal introduction of PINNs can be traced back to the work of Raissi, Perdikaris, and Karniadakis in 2019. Their groundbreaking paper, "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations," laid the foundation for this field. They demonstrated that by embedding the residuals of PDEs into the loss function of neural networks, it is possible to obtain solutions that are not only data-driven but also physically consistent.

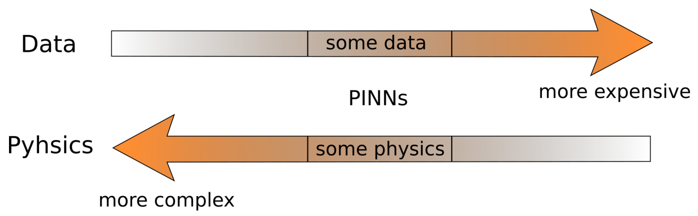


Figure 3.1: Trade-off between data and physics

As depicted in Figure 3.1, obtaining more data is more expensive due to the need for more simulation or experiments, while involving more physics increases the complexity of the problem. Herein, physics-informed neural networks (PINNs) come into play, which requires some data and some physics. PINNs enforce the neural network architecture to learn the pattern by using the limited data with the help of the governing equations of the problem.

3.1 Formulation of a PINN

Consider a partial differential equation as

$$\frac{\partial u}{\partial t} + \mathcal{N}[u; \lambda] = 0, x \in \Omega, t \in [0, \mathcal{T}] \tag{3.1}$$

where $u(t, x)$ is the latent solution with time $t \in [0, \mathcal{T}]$ and a spatial variable $x \in \Omega$, $\mathcal{N}[u; \lambda]$ is the nonlinear differential operator with the coefficient λ, and Ω is a subset of $\mathbb{R}^D$.

The conventional methods to solve Eq. 3.1 are either analytical or numerical. The analytical solutions are not always feasible and numerical methods are expensive. The PINNs approach can be considered more feasible than analytical methods and less expensive than numerical methods. The physics-informed neural network estimates the solution $u(t, x)$ and the physics-enhanced part evaluates the given partial differential equation using the estimated solution. As shown in Fig. 3.2, the physics-informed neural network predicts the solution $u(x, t)$ by using fully connected layers with nonlinear activation functions, denoted as σ. The physics-enhanced part evaluates the chosen PDE, denoted as $f(x)$, using the higher derivatives of the solution $u(x, t)$, denoted here d_x, d_{xx} and d_t, with respect to space (x) and time (t), and the identity matrix of u. Since the solution is known on the boundaries and the approximated solution must fulfill the governing PDE, the loss can be computed using a cost function, i.e. mean squared error. If the solution is not converged, the weights of the neural network are updated using back-propagation to estimate the new solution, until the given convergence limit is reached.

Compared to classical neural networks, the loss function of a PINN has three terms:

$$L = E_u + E_0 + E_f \tag{3.2}$$

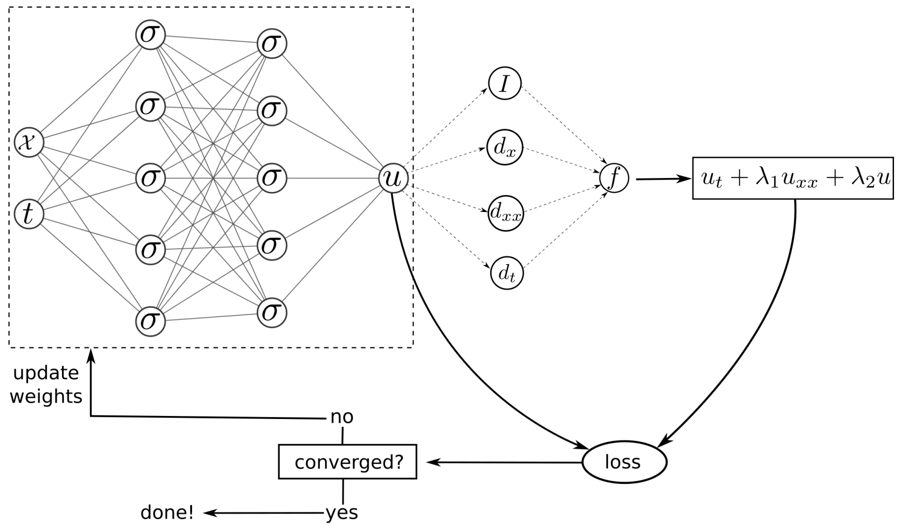


Figure 3.2: The visionary representation of physics-informed neural network for an arbitrary space-time dependent partial differential equation.

The first term, denoted as E_u, calculates the error for the approximated solution on the known boundaries. In the case of a one dimensional non-linear heat equation subjected to the following homogenous Neumann boundary condition

$$\frac{\partial u(x=1,t)}{\partial x} = h, \tag{3.3}$$

the Dirichlet boundary condition

$$u(x=0,t) = g, \tag{3.4}$$

E_u term is calculated as

$$E_u = E_{Neumann} + E_{Dirichlet}, \tag{3.5}$$

where

$$E_{\text{Neumann}} = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(\frac{\partial}{\partial x} u_P \left(x_b^i, t_b^i\right) - h \right)^2 \tag{3.6}$$

enforces the homogenous Neumann boundary condition (refer to 3.3) by penalizing the error between the derivative of the predicted solution, denoted

as $\frac{\partial}{\partial x} u_P\left(x_b^i, t_b^i\right)$, and the given Neumann boundary condition h at N_b random points $\left\{x_b^i, t_b\right\}_{i=1}^{N_b}$ on the boundary $x_b = 1$.
The term, $E_{Dirichlet}$ of the Eq. 3.5

$$E_{\text{Dirichlet}} = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(u_P\left(x_b^i, t_b^i\right) - g\right)^2 \tag{3.7}$$

enforces the Dirichlet boundary condition according Eq. 3.4 by penalizing the error between approximated solution $u_P(x_b^i, t_b^i)$ and prescribed Dirichlet boundary condition g at N_b random points $\left\{x_b^i, t_b\right\}_{i=1}^{N_b}$ on the boundary $x_b = 0$.

The initial condition loss, E_0 of the Eq. 3.2

$$E_0 = \frac{1}{N_0} \sum_{i=1}^{N_0} \left(u_P\left(x_0^i, 0\right) - u_0^i\right)^2, \tag{3.8}$$

with the following initial condition

$$u(x, t = 0) = u_0, \tag{3.9}$$

enforces the initial conditions at N_0 random points $\left\{x_0^i, t_b\right\}_{i=1}^{N_0}$ at initial time $t_b = 0$

The loss functions E_u and E_0 contain only the loss on the known boundaries and on the known initial conditions, respectively.

On the other hand, the final term of the Eq. 3.2

$$E_{\text{f}} = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(\frac{\partial}{\partial t} u_P\left(x_f^i, t_b^i\right) - \mathcal{N}[u_P(x_b^i, t_b^i)]\right)^2 \tag{3.10}$$

imposes the given partial differential equation at every random collocation point inside the domain by penalizing the estimated left-hand side and the known right-hand side of the governing equation. Since PINNs are mesh-free, the distribution of collocation points can be uniform or random.

After calculation of the loss terms, weights of the architecture is updated using back-propagation with the help of a optimization function, i.e. stochastic gradient descent, Adam etc.

3.2 Comparison between PINNs and FEM

To further explain the ideas of PINNs and to help those with knowledge of FEM to understand PINNs more easily, we make a comparison between PINNs and FEM point by poin

	PINN	**FEM**
Basis function	Neural network (nonlinear)	Piecewise polynomial (linear)
Parameters	Weights and biases	Point values
Training points	Scattered points (mesh-free)	Mesh points
PDE embedding	Loss function	Algebraic system
Parameter solver	Gradient-based optimizer	Linear solver
Errors	$\mathcal{E}_{\text{app}}$:Approximation Error, $\mathcal{E}_{\text{gen}}$, and $\mathcal{E}_{\text{opt}}$	Approximation/quadrature errors
Error bounds	Not available yet	Partially available

Table 3.1: Comparison between PINN and FEM.

- Approximation Error ($\mathcal{E}_{\text{app}}$): This error occurs because the neural network might not be complex enough to perfectly model the true solution of the differential equation. It is influenced by the architecture of the network, including the number of layers, neurons per layer, and the types of activation functions used. A simple neural network might not capture the intricate details of the solution, leading to this type of error. To reduce approximation error, you can increase the complexity of the neural network by adding more layers or neurons, or by employing more sophisticated architectures that can better represent the underlying solution.

- Generalization Error ($\mathcal{E}_{\text{gen}}$): which measures how well the trained model performs on new, unseen data. This error often arises from overfitting, where the model learns the noise and specific details of the training data instead of the general trend. Overfitting results in a model that performs well on the training data but poorly on new data. To address generalization error, techniques such as cross-validation, regularization methods (like dropout or L2 regularization), and maintaining a separate validation dataset during training can be employed. These strategies help ensure that the model captures the true underlying patterns rather than the noise in the training data.

- Optimization Error ($\mathcal{E}_{\text{opt}}$) : occurs when the optimization algorithm fails to find the global minimum of the loss function. This error can be due to several factors, including the choice of optimization algorithm, learning rate, and other hyperparameters. The optimization process might get stuck in local minima or saddle points, leading to suboptimal solutions. To minimize optimization error, one can use advanced optimization techniques such as the Adam optimizer, adjust learning rate schedules, and ensure proper initialization of the neural network weights. Fine-tuning these hyperparameters can significantly improve the chances of finding a better solution.

3.3 General remarks regarding PINNs

- A PINN is mesh-free so it can handle unstructured or irregular domains.

- PINNs can be deployed on both forward and inverse problems. In forward problems, they can find the hidden solution $u(x,t)$ where the coefficients λ are known (Eq. 3.1) or discover the parameters λ using the provided solution, which are known as inverse problems. This approach is called also as data-driven identification.

- Depending on the type of the governing equation, they can be used on a wide variety of problems ranging from biological systems (reconstruction clinical magnetic resonance imaging (MRI) data) to the governing equations of continuum mechanics.

- Since the governing equations of the problem are involved in the learning process, PINNs can learn even data is limited.

- The derivatives of the network prediction u_P with respect to space (x) and time (t) can be easily and efficiently computed by the automatic differentiation capabilities implemented in many deep learning tool kits, i.e. Tensorflow and Pytorch .

3.4 DeepDXE library

DeepXDE (Deep Learning for Differential Equations) is a Python library designed to solve differential equations using deep learning techniques, and it's open-source for anyone to use. Whether you're dealing with ordinary differential equations (ODEs), partial differential equations (PDEs), or integro-differential equations, DeepXDE supports a wide variety of them. One of

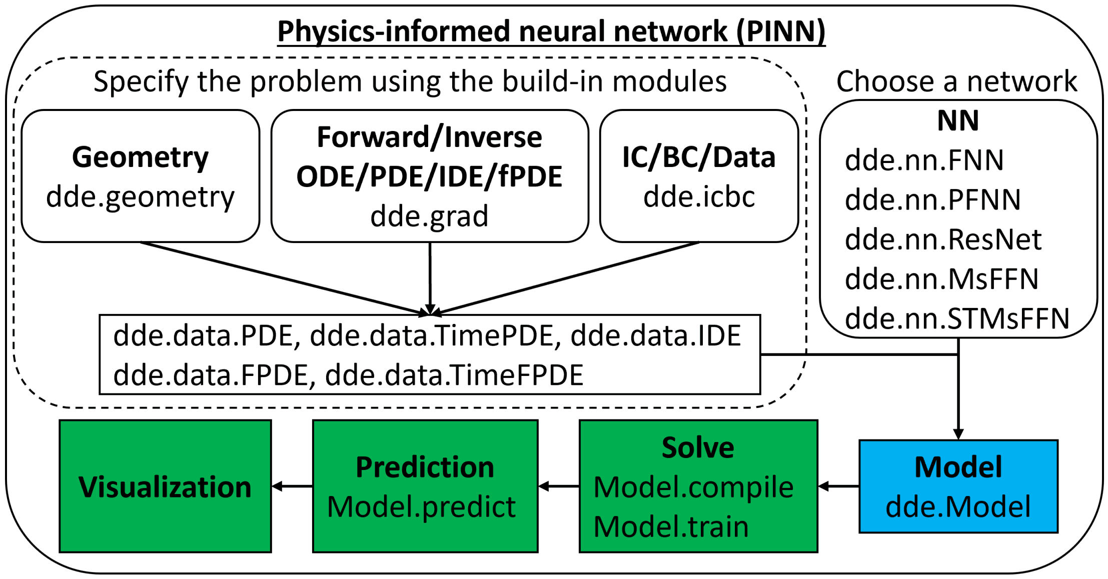


Figure 3.3: Diagram illustrating the architecture of DeepXDE

its standout features is its user-friendly interface, which makes it easy for researchers and practitioners across different fields to define and solve problems using Physics-Informed Neural Networks (PINNs). You can specify your differential equations, boundary conditions, and initial conditions with ease, making it adaptable for diverse problems in various domains.

DeepXDE also supports different types of neural network architectures like fully connected networks, recurrent networks (RNNs), and convolutional networks (CNNs). This flexibility lets you choose the right architecture depending on the complexity of your problem.

Built on TensorFlow, a popular deep learning framework, DeepXDE ensures compatibility with TensorFlow's extensive features and ecosystem. It uses automatic differentiation to compute derivatives needed for solving differential equations, which reduces manual effort and potential errors.

Additionally, DeepXDE provides visualization tools that help you see and understand the solutions to your differential equations. These tools aid in interpreting and analyzing results, making it a comprehensive toolset for tackling challenging differential equation problems with deep learning.

Algorithm to solve PDEs using DeepXDE

Problem Definition:

Specify the physical problem involving a PDE and its boundary conditions.

Initialization:

Import necessary libraries for numerical computation and deep learning frameworks if applicable.

Define Helper Functions:

Implement functions to compute derivatives and define physical parameters relevant to the problem.

Define the PDE:

Express the PDE in terms of its derivatives and any external forces or coefficients affecting the system.

Boundary Conditions:

Define boundary conditions that specify the behavior of the solution at the domain boundaries.

Define the Exact Solution (Optional):

If available, specify an exact solution to validate the numerical solution and monitor convergence.

Geometry and Boundary Conditions:

Define the geometric domain and set up boundary conditions using appropriate classes or functions provided by the computational framework.

Generate PDE Data:

Generate training and testing data that satisfy the PDE and boundary conditions, ensuring coverage of the domain.

Neural Network Architecture:

Define the neural network architecture suitable for solving the PDE, considering layer sizes, activation functions, and initialization methods.

Model Compilation and Training:

Compile the model, specifying the optimizer, learning rate, and metrics for evaluation.

Train the model over multiple epochs, monitoring performance metrics and displaying progress as required.

Post-Training Analysis:

Optionally, save or visualize training metrics and model state to analyze convergence and performance.

Chapter 4

The Euler-Bernoulli Beam Theory with Governing Equations

A beam is a structural element having one dimension that is much larger than the other two and capable of resisting the loads applied laterally to the beam axis. Since the further investigations are pursued for the Euler-Bernoulli beam theory, known as thin beam theory, the main assumptions and the governing equations of the Euler-Bernoulli are given in more detail.

4.1 Assumptions of the Euler-Bernoulli Beams

The main hypothesis of the Euler-Bernoulli beam theory is that plane cross-sections which are orthogonal to the beam axis (neutral axis) stay still plane and orthogonal to the beam axis after deformation. The overall assumptions of the Euler beam theory:

- Plane cross-sections remain plane and perpendicular to the neutral axis of the beam after deformation.
- The deformations are small, so the equilibrium is computed on the reference configuration (Small deformation theory).
- Behavior of the beam is linear elastic and isotropic.
- Shear strains are neglected.

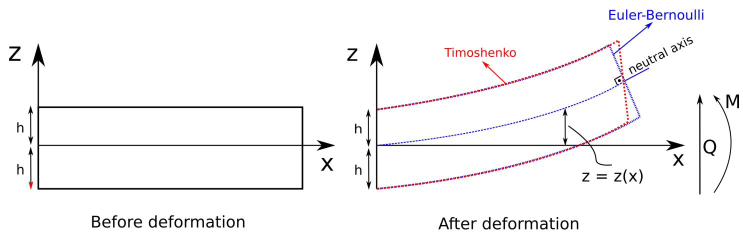


Figure 4.1: The Euler-Bernoulli and Timoshenko beam deflections

As depicted in Fig. 4.1, the plane cross-sections stay plane and perpendicular to the beam axis in the Euler-Bernoulli beam. In the Timoshenko beam, plane cross-sections still remain plane but are no longer perpendicular to the beam axis. The difference between normal to the beam axis and plane cross-section is known as shear deformation. Here, shear strains will be ignored so the beam Formulation is considered under the Euler-Bernoulli beam theory, which is known also as thin beam theory.

4.2 Governing Equations

Static beam equation

The governing ordinary differential equation is obtained for a static model

$$\frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 w(x)}{\partial x^2}\right) + q(x) = 0, x \in \Omega, \tag{4.1}$$

and, similarly if $q(x)$ is a concentrated load

$$q(x) = P\delta\left(x - x_0\right). \tag{4.2}$$

4.3 Boundary Conditions and Beam Supports

To estimate the unknown solution $w(x,t)$, the governing equation of the Euler-Bernoulli beam must be solved using the initial and boundary conditions. Since the governing equation is of the $4th$ order in space , four different boundary conditions are introduced:

- $w(x,t) \rightarrow$ represents the displacement in the z direction,
- $w_x(x,t) \rightarrow$ indicates the slope of the beam,

- $w_{xx}(x,t) \rightarrow$ measures the bending moment,
- $w_{xxx}(x,t) \rightarrow$ measures the shear force,

at location $x \in \Omega$ and time $t \in [0,T]$.

Similary, and two initial conditions are needed

- $w(x,0) \rightarrow$ represents the displacement in the z direction,
- $w_t(x,0) \rightarrow$ indicates the time derivative of the displacement,

at location $x \in \Omega$ and at time $t = 0$. Initial conditions are not taken into account in the static beam equation since the time-dependent derivative term vanishes.

Two main supports will be further investigated:

- fixed supported beams
- simply supported beams

4.3.1 Cantilever beam

A cantilever beam is fixed supported at one end and free at the other as depicted in Fig. 4.2. No displacement and rotation are allowed at the fixed edge. No bending moment and no shear force at the free edge are generated.

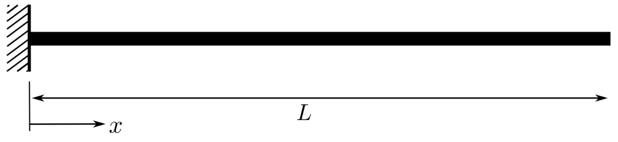


Figure 4.2: An illustration of a fixed supported beam

Boundary conditions of a cantilever beam

$$\begin{array}{ll} w(x,t) = 0 \quad and \quad w_x(x,t) = 0, & at\ x = 0, t = [0,T] \\ w_{xx}(x,t) = 0 \quad and \quad w_{xxx}(x,t) = 0, & at\ x = L, t = [0,T] \end{array} \tag{4.3}$$

4.3.2 Simply supported beam

Simply supported beam is pinned at the ends for two edges as shown in Fig. 4.3. At both pinned ends, no displacement but rotation is allowed which means that no bending moments are generated.

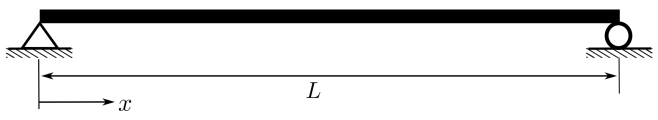


Figure 4.3: An illustration of a simply supported beam.

Boundary conditions of a simply supported beam

$$\begin{array}{lll} w(x,t)=0 \quad \textit{and} \quad w_{xx}(x,t)=0, & \textit{at } x=0, t=[0,T] \\ w(x,t)=0 \quad \textit{and} \quad w_{xx}(x,t)=0, & \textit{at } x=L, t=[0,T] \end{array} \tag{4.4}$$

4.3.3 Clamped beam

A clamped beam is fixed supported at both ends as depicted in Fig. 4.4. At both fixed ends, no displacement and rotation is allowed.

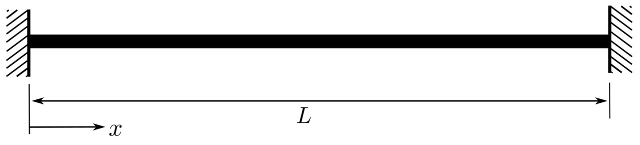


Figure 4.4: An illustration of a clamped beam.

Boundary conditions of a clamped beam

$$\begin{array}{lll} w(x,t)=0 \quad \textit{and} \quad w_{x}(x,t)=0, & \textit{at } x=0, t=[0,T] \\ w(x,t)=0 \quad \textit{and} \quad w_{x}(x,t)=0, & \textit{at } x=L, t=[0,T] \end{array} \tag{4.5}$$

Chapter 5

Application of PINNs on 1D Beams

Here, one-dimensional Euler-Bernoulli beams under the distributed load with uniform geomerical and material properties are conducted, which are governed by an ordinary differential equation (Eq. 4.1) in static models

5.1 Static Models

Consider the governing equations of one-dimesional linear elastic Euler-Bernoulli beam (Fig. 4.4) with the corresponding boundary conditions of clamped beam as

$$\frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 w(x)}{\partial x^2}\right) + q(x) = 0 \quad \text{on } \Omega \tag{5.1}$$

$$\frac{\partial w(x)}{\partial x} = h \quad \text{on } \Gamma_N, \tag{5.2}$$

$$w(x) = g \quad \text{on } \Gamma_D. \tag{5.3}$$

The Neumann and Dirichlet boundary conditions are denoted as Γ_N and Γ_D. To give a better understanding, the spatial domain is defined $\Omega = [0, L]$, with $\Gamma_D = \{x|x = 0, x = L\}$ and $\Gamma_N = \{x|x = 0, x = L\}$. If the length of the beam is set to $L = 1$, the following boundary conditions of a clamped support Euler-Bernoulli beam are obtained

$$\begin{aligned} w(x = 0) = 0 \text{ and } \quad w_x(x = 0) = 0, \\ w(x = 1) = 0 \text{ and } \quad w_x(x = 1) = 0. \end{aligned} \tag{5.4}$$

The physics-informed neural network approximates the unknown solution $w(x)$ through the hidden layers enhanced with non-linear activation functions σ using the spatial domain x as the input as depicted in Fig. 5.1. Since only the solution on the boundaries are known, the predicted solution for every location inside the domain must fulfill the Eq. 5.1 such that

$$f := \frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 w(x)}{\partial x^2}\right) + q(x). \tag{5.5}$$

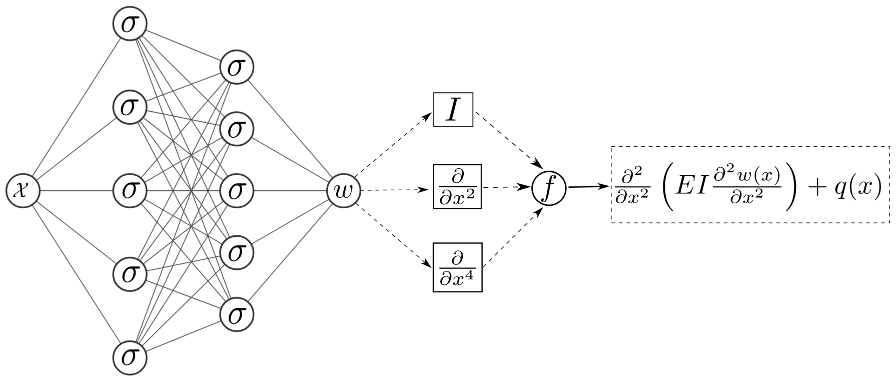
Here the f term represents the residuum or physics-informed residual which has to be fulfilled at every point inside the domain. The main purpose of PINNs is to minize the residuum enforcing the boundary conditions via the loss function.

The loss function of static clamped Euler-Bernoulli beam is defined

$$\begin{aligned} L &= E_u + E_f \\ &= E_{Neumann} + E_{Dirichlet} + E_f \\ &= \frac{1}{N_b}\sum_{i=1}^{N_b}\left[\left(\frac{\partial}{\partial x}w_P\left(x_b^i\right) - h\right)^2 + \left(w_P\left(x_b^i\right) - g\right)^2\right] + \frac{1}{N_f}\sum_{i=1}^{N_f}\left(f_P(x_f^i)\right)^2 \\ &= \frac{1}{N_b}\sum_{i=1}^{N_b}\left[\underbrace{\left(\frac{\partial}{\partial x}w_P\left(x_b^i\right)\right)^2}_{\text{Neumann term}} + \underbrace{\left(w_P\left(x_b^i\right)\right)^2}_{\text{Dirichlet term}}\right] + \frac{1}{N_f}\sum_{i=1}^{N_f}\left(f_P(x_f^i)\right)^2 \end{aligned} \tag{5.6}$$

The h and g terms are the prescribed Neumann and Dirichlet boundary conditions at N_b boundary points $\{x_b^i\}_{i=1}^{N_b} = \{0, 1\}$. Since the beam is considered as a clamped beam, $h = 0$ and $g = 0$. The last term of Eq. 5.6 minizes the error at every point inside the spatial domain, called as the collocation points $\left\{x_f^i\right\}_{i=1}^{N_f}$. The collocation points can be generated using random or uniform distributions.

For demonstrating the application of PINNs on the Euler-Bernoulli beams, three cases are further investigated to involve the higher-order derivatives in the loss function as depicted in Fig. 5.2. For instance, the clamped beam contains only the first order, while the cantilever beam contains the first, second and third-order derivatives (cf. 4.2, 4.3, 4.4). For simplicity, Young's modulus and the moment of inertia terms are set to $AE = 1$, the beam length and the magnitude of the distributed loading are set to $L = 1$ and $q = 1$, relatively.



σ
x
w
I
$\frac{\partial}{\partial x^2}$
$\frac{\partial}{\partial x^4}$
f
$\frac{\partial^2}{\partial x^2}\left(EI\frac{\partial^2 w(x)}{\partial x^2}\right) + q(x)$

Figure 5.1: An illustration of physics-informed neural network for the governing ordinary differential equation of the one-dimensional Euler-Bernoulli beam.

The following PINN architecture is used to compute the approximated solution:

- Architecture contains 1 input layer, 1 output layer and 3 hidden layers containing 30 nodes.
- Hyperbolic tangent activation functions *tanh* are used to non-linearize the outputs of the layers.
- The *Glorot Uniform* initializer is applied to initialize the weights randomly to avoid dying out of the nodes.
- The *Adam* optimizer with learning rate *0.0005* is chosen as the optimization function.
- The number of collocation points is set $N_f = 20$ and the Sobol sampling is performed to have a non-uniform distribution.

The results of the investigated cases are given in Fig. 5.3. Figures on the left show that the predicted displacements are almost identical to the analytical solutions at each collocation point as can be seen in the magnitude of the loss function. On the other hand, there is a difference in terms of the order of magnitudes of the loss functions between the differential equation loss and the total boundary losses. The reason is that the boundary conditions are not explicitly enforced. In Eq. 5.6, the boundary condition losses are added to overall lost and this overall loss is minimized. To enforce

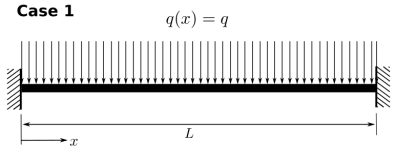
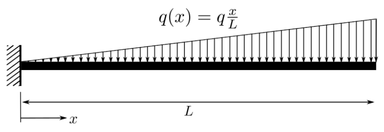
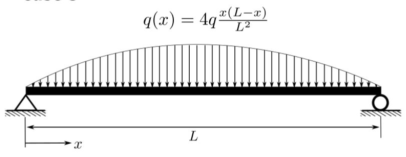
Beam model	**Description**
Case 1 	Clamped beam under a uniformly distributed load q(x) with the magnitude q. The analytical solution: $w(x) = -\frac{qx^2}{24EI}(x-L)^2$
Case 2 	Cantilever beam under a triangular distributed load q(x) with the magnitude q at x=L. The analytical solution: $w(x) = -\frac{qx^2}{120EI}(20L^3 - 10L^2x + x^3)$
Case 3 	Simply supported beam under a quadratically distributed load q(x) with the magnitude q at x=L/2. The analytical solution: $w(x) = -\frac{qx}{90EI}(x^5 - 6x^4L + 15x^3L^2 - 15x^2L^3 + 5L^5)$

Figure 5.2: Three investigated beam models with their descriptions and analytical solutions.

the boundary conditions, the loss function can be reformulated to a soft constraint problem or penalty methods can be considered.

Here is the link to my Python file click Here.

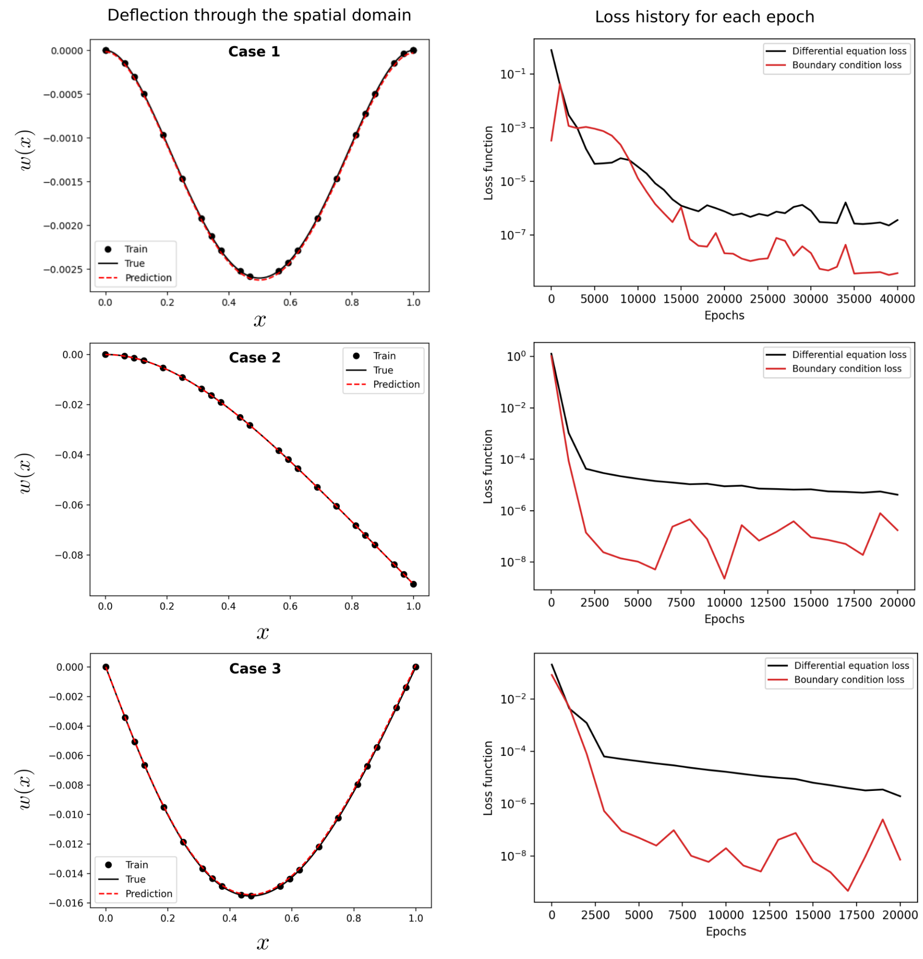


Figure 5.3: On the figures left, the predicted and true deflections through the spatial domain are compared for three different cases. On the figures right, the loss history containing the differential equation and boundary condition losses are given for each epoch.

Conclusion

In this report, we've explored how Physics-Informed Neural Networks (PINNs) can solve partial differential equations (PDEs), focusing specifically on their application to beam static equations. By incorporating the laws of physics directly into their framework, PINNs offer a novel and efficient way to solve these complex equations without relying solely on traditional numerical methods.

Our study demonstrated that PINNs excel in handling the static equations governing beam mechanics. Unlike conventional methods that often involve labor-intensive steps like discretization and mesh generation, PINNs seamlessly integrate the essential physical principles of beam theory. This leads to solutions that are not only accurate but also computationally efficient, streamlining the entire process.

Looking ahead,scientific machine learning is emerging as a new and potentially powerful alternative to classical scientific computing, so a libraries such as DeepXDE will accelerate this development and make it accessible not only to stu- dents but also to other researchers who may find the need to adopt PINN-like methods in their research, which can be very effective, especially for inverse problems

In conclusion, our exploration of PINNs for solving beam static equations highlights their significant potential in civil engineering. But the technology is not without limitations. PINNs still have some limitations. For forward problems, PINNs are currently slower than finite elements, but this can be alleviated via offline training. For long time integration, one can also use time-parallel methods to simultaneously compute on multiple GPUs for shorter time domains. Another limitation is the search for effective neural network architectures, which is currently done empirically by users; however, emerging meta-learning tech- niques can be used to automate this search; see . Moreover, while here we enforce the strong form of PDEs, which is easily implemented using automatic dif- ferentiation, alternative weak/variational forms may also be effective, although they require the use of quadrature grids. Many other extensions for multiphysics and multiscale problems are possible across different scientific disciplines by creatively de- signing the loss function and introducing suitable solution spaces. For instance, in the five examples we present here, we only assume data on scattered points; however, in geophysics or biomedicine we may have mixed data in the form of images and point measurements. In this case, we can design a composite neural network consisting of one convolutional neural network and one PINN sharing the same set of parameters, and minimize the total loss which could be a weighted summation of multiple losses from each neural network.

Bibliography

[1] IMCS-COMPSIM. (n.d.). PINNs for Computational Mechanics. Retrieved from `https://github.com/imcs-compsim/pinns_for_comp_mech`

[2] Raissi, M., Perdikaris, P., Karniadakis, G. E. (2019). Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations. *SIAM Journal on Scientific Computing*, 41(1), A1-A30. Retrieved from `https://epubs.siam.org/doi/epdf/10.1137/19M1274067`

[3] Brownlee, J. (2019). Using Activation Functions in Neural Networks. Retrieved from `https://machinelearningmastery.com/using-activation-functions-in-neural-networks/`

[4] Brownlee, J. (2019). Application of Differentiations in Neural Networks. Retrieved from `https://machinelearningmastery.com/application-of-differentiations-in-neural-networks/`

[5] Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Retrieved from `http://neuralnetworksanddeeplearning.com` Chapter 2: The Four Fundamental Equations Behind Backpropagation. Retrieved from `http://neuralnetworksanddeeplearning.com/chap2.html#the_four_fundamental_equations_behind_backpropagation`

[6] Brunton, S. L., Kutz, J. N. (2019). *Deep Learning in Computational Mechanics: An Introductory Course.*

[7] UCLA LabX. (2020). Chapter 7: Regularization. In *Deep Learning Book Notes.* Retrieved from `https://ucla-labx.github.io/deeplearningbook-notes/Ch7-Regularization.html`