An Educational Demonstration Of Using Neural Networks To Solve Partial Differential Equations

Jacky Song

April 2025

Abstract

Physics-informed neural networks (PINNs) [10] offer an alternative paradigm to solving partial differential equations (PDEs). Using methods developed for machine learning, PINNs have shown significant success in finding numerical solutions to PDEs at a fraction of the computational complexity of traditional methods.[2] This paper does not intend to advance the state-of-the-art in the field, but rather provides a useful and simple demonstration of a PINN for educational purposes. It is hoped that this paper may give a simple educational framework for introducing PINNs to an unfamiliar audience.

1 Introduction

Partial differential equations (PDEs) are key to our understanding of natural phenomena, from the weather to ocean waves, to aerodynamics and beyond. However, it is well-known that most boundary-value problems (BVP) for PDEs have no analytical solution, and the solution must be approximated numerically. Mesh-based numerical methods for PDEs, like finite-difference and finite-element methods, have long been the mainstay of computational physics and engineering. However, mesh-based methods suffer from a variety of fundamental limitations, including difficulty in obtaining a solution with irregular geometries and worsening performance with increased dimensionality [9].

For this reason, there is great interest in developing meshfree methods for solving PDEs. Raissi et al. introduced the notion that using neural networks as function approximators could serve as a meshfree method for solving PDEs [10]. They showed that by converting a PDE into an optimization problem, one could use a neural network to numerically-approximate the solution to the PDE, a paradigm known as *physics-informed neural networks* (PINNs).

A basic overview of the essential mechanism of a PINN is now given. Neural networks are generalized, differentiable mathematical models that predict an output variable $\mathbf{y}$ from given input variables $\mathbf{x}$. While the specific mathematical structure of a neural network may greatly vary, it is instructive to examine one

of the simplest models, the so-called *multi-layer perceptron* (MLP), from which many PINNs are derived[8].

An MLP consists of a composition of functions $\mathbf{f}_i(\mathbf{x})$, each of which is known as a *layer*. Thus it takes the mathematical form:

$$\mathbf{F}(\mathbf{x}) = (\mathbf{f}_1 \circ \mathbf{f}_2 \circ \mathbf{f}_3 \cdots \circ \mathbf{f}_{n-1} \circ \mathbf{f}_n)(\mathbf{x}) \tag{1}$$

Each layer is a vector-valued function of weights w_{mn} and biases b_n, known as the *parameters* of the layer. With only the weights and biases, a layer would be a linear function, unable to describe nonlinear relationships between the input and output variables. Thus, one may choose to include a nonlinear *activation function* $\sigma(\mathbf{x})$. The layer may then be mathematically-represented as:

$$\mathbf{f}_i(\mathbf{x}) = \sigma\left(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix}_i \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_i + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}_i\right) \tag{2}$$

Or, in perhaps more familiar-looking notation reminiscent of linear regression:

$$\mathbf{f}_i(\mathbf{x}) = \sigma(w_{mn}x_n + b_n) = \sigma(\mathbf{w}\mathbf{x} + \mathbf{b}) \tag{3}$$

Upon initializing the model, the weights are set to random numbers, and the biases are set to zero. The model then uses an optimization algorithm to "train" itself to predict the correct outputs $\mathbf{y}$ given inputs $\mathbf{x}$. First, one defines a *loss function* that measures the difference of a model's predictions (that is, the value of $\mathbf{F}(\mathbf{x})$) from the true outputs (that is, $\mathbf{y}$). A common loss function is the *mean-squared error* (MSE) loss, given by:

$$L(\mathbf{x}, \mathbf{F}(\mathbf{x}), \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}\left([\mathbf{y} - \mathbf{F}(\mathbf{x})]^2\right)_i \tag{4}$$

By minimizing the loss function, the error in the model's predictions compared to the true values $\mathbf{y}$ are also minimized. Thus, "training" the model is essentially the *optimization problem* of minimizing the loss function[11]. In the simplest gradient-based optimization scheme, after every iteration i, one updates the weights and biases based on the gradient of the loss function, scaled by a constant ϵ that controls the strength of the optimizer:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \epsilon\nabla_{\mathbf{w}_i}L \tag{5}$$

$$\mathbf{b}_{i+1} = \mathbf{b}_i - \epsilon\nabla_{\mathbf{b}_i}L \tag{6}$$

As the optimizer is run repeatedly, the loss is incrementally reduced, and the model becomes more accurate; once the model is deemed to predict with satisfactory accuracy, it is said to be *trained*. The field of devising, building, and testing such types of self-optimizing models (though not exclusive to

neural networks) is known as *machine learning*. Sophisticated research in the field has optimized nearly every part of the process, using innovations such as reverse-mode automatic differentiation to calculate gradients down to machine precision[1] (among others), and developing highly-efficient optimizers such as the popular Adam optimizer [6].

The generalizable nature of neural networks, including the MLP, have contributed to their enduring popularity and widespread use. By the universal approximation theorem, one may approximate essentially any smooth function with a neural network [4]. In PINNs, this property is utilized to transfer machine-learning techniques to the domain of solving PDEs.

Consider a neural network model $\hat{u}(\mathbf{x}, t)$ (for instance, an MLP) that approximates the solution $u(\mathbf{x}, t)$ to a particular scalar-valued PDE of the form:

$$G\left(u, t, x, y, \dots \frac{\partial u}{\partial t}, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \dots \frac{\partial^2 u}{\partial t^2}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}\right) = 0 \tag{7}$$

The PDE is defined over domain Ω with boundary B, and has appropriate boundary and initial conditions, given by:

$$u(\mathbf{x}, t = 0) = u_0(\mathbf{x}) \tag{8}$$
$$\nabla u(\mathbf{x}, t = 0) = D_0(\mathbf{x}) \tag{9}$$
$$u(x, t)\Big|_{\mathbf{x}\in B} = \mathrm{BC}_1(\mathbf{x}, t) \tag{10}$$
$$\nabla u(x, t)\Big|_{\mathbf{x}\in B} = \mathrm{BC}_2(\mathbf{x}, t) \tag{11}$$
$$\vdots \tag{12}$$
$$\nabla^2 u(x, t)\Big|_{\mathbf{x}\in B} = \mathrm{BC}_n(\mathbf{x}, t) \tag{13}$$

By definition, the correct solution to the PDE must be one that makes $G(u, \nabla u, \nabla^2 u, t)$ zero, so a good approximate solution must be one that *minimizes* G. Likewise, a good approximate solution must approximately satisfy the initial and boundary conditions, or, in other words, minimize the *difference* between the true initial/boundary conditions and the predicted values of the model $\hat{u}$ at the initial time and on boundary points of the domain.

This motivates casting the problem into an *optimization problem*, just like in machine learning. This may be done by defining a custom loss function, given by:

$$\begin{aligned} L(\mathbf{w}, \mathbf{b}; x, t) = {} & \frac{1}{n}\sum_{i=1}^{n}\left(G(\hat{u}, \mathbf{x}, t, \nabla\hat{u}, \nabla^2\hat{u})^2\right)_i + \frac{1}{n}\sum_{i=1}^{n}\left([\hat{u}(\mathbf{x}, 0) - u_0(\mathbf{x})]^2\right)_i \\ & + \frac{1}{n}\sum_{i=1}^{n}\left((\nabla\hat{u}(\mathbf{x}, 0) - D_0(\mathbf{x}))^2\right)_i + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\left(u(\mathbf{x}, t)\Big|_{\mathbf{x}\in B} - \mathrm{BC}_j(\mathbf{x}, t)\right)^2\right)_i \end{aligned}$$

One then proceeds to train the neural network in much the same way as a conventional neural network. That is to say, a trial solution is approximated by the neural net, and using gradient-based optimization, the loss is minimized until the neural network converges to the true solution.

2 An educational demonstration

An simple, open-source demonstration of two PINNs was created as part of this research, and has been made publicly-available (more details in section 7). The first demonstration used the following initial-boundary value problem for the one-dimensional wave equation, $u_{tt} = c^2 u_{xx}$:

$$u(x, 0) = Ae^{-\beta x^2} \tag{14}$$

$$\frac{\partial u}{\partial t}(x, 0) = 0 \tag{15}$$

$$u(-\infty, t) = u(\infty, t) = 0 \tag{16}$$

$$x \in (-\infty, \infty) \tag{17}$$

The analytical solution to the above problem is well-known to be given by:

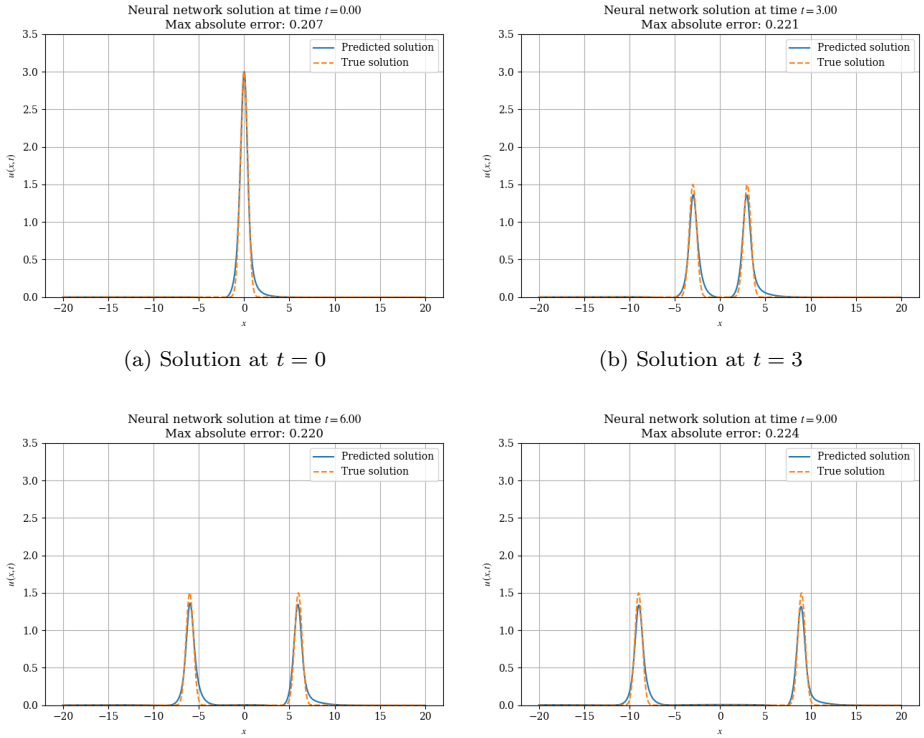
$$u(x, t) = \frac{A}{2}[e^{-\beta(x-ct)^2} + e^{-\beta(x+ct)^2}] \tag{18}$$

Meanwhile, the second demonstration used a nonlinear variant of the one-dimensional wave equation, given by $u_{tt} = c^2 u_{xx} + \varepsilon u^2$, where ε is a constant controlling the strength of the nonlinearity. This nonlinear wave equation was solved for the same initial and boundary values as in the first demonstration. However, it has no analytical solution, so the solution can only be numerically-approximated.

3 Results and analysis

The open-source `PINA` Python library, which implements the core functions and models for developing PINNs, was used for creating the solvers for the demonstrations. Both neural networks were multi-layer perceptrons with the custom PINN loss function. For the first demonstration, the parameters $c = 1, A = 3, \beta = 3$ were used, where the solver domain was $x \in [-20, 20], t \in [0, 10]$. The approximation $\lim_{|x|\to 20} u(x, t) \approx u(\pm\infty, t) = 0$ was used as solving the wave equation over the entire real line is clearly not possible. This is a sound approximation because the Gaussian wave profile decays rapidly to zero, so a large domain is not needed.

The MLP used had two layers and 501 parameters (weights and biases) in total. The domain was sampled with 8000 points and the neural network was trained for 3000 iterations. The results are shown in the below figure:



(a) Solution at $t = 0$

(b) Solution at $t = 3$

(c) Solution at $t = 6$

(d) Solution at $t = 9$

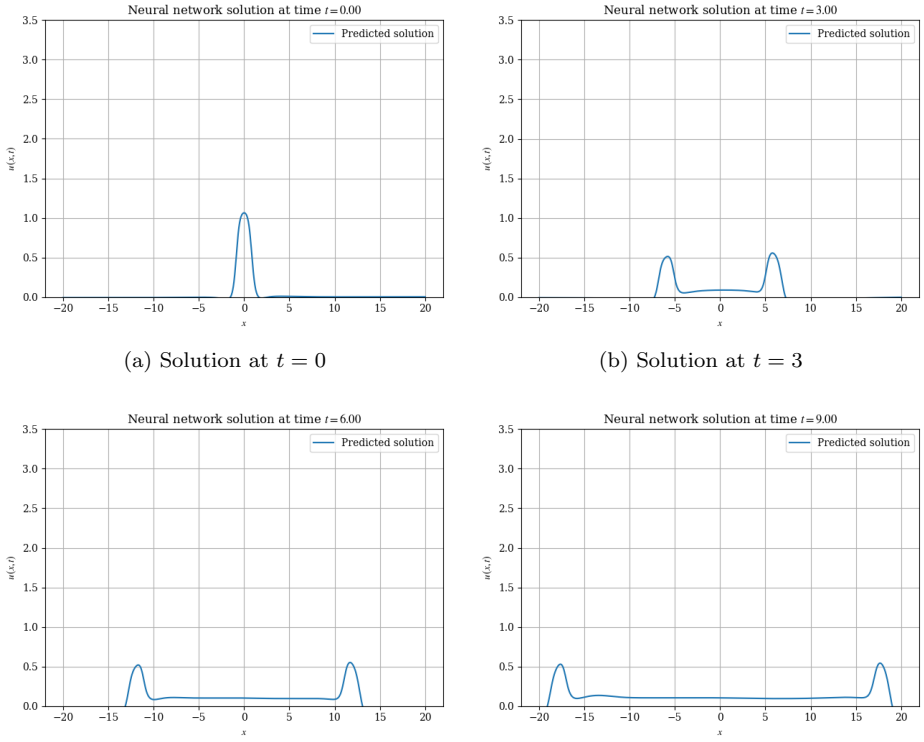
Figure 1: Neural network-predicted solution for linear wave equation

In the first demonstration, the neural network's predicted solution closely matched the analytical solution, albeit still with some error. While the total error remained fairly low, it increased with time, becoming about 8% higher by $t = 9$ compared to its value at $t = 0$. In addition, a fairly large number of iterations was required for satisfactory convergence of the solution, which is a challenge for all neural network-based models in general.

Meanwhile, for the nonlinear case, the parameters used were $c = 2, A = 1, \beta = 3, \varepsilon = 0.5$. Lower values of ε were found to exhibit nonlinear effects that were too weak to be easily noticed, while higher values resulted in nonsensical solutions, so a moderate value was therefore chosen. As the quadratic dependence on u renders the solution numerically-unstable, the neural network was trained over the time interval $t \in [0, 5]$ only, although it was evaluated over the full $t \in [0, 10]$ interval.

For solving the nonlinear wave equation, a more complex MLP was used, with three layers and 2,900 parameters in total. The number of sampled points in the domain was increased to 10,000 for greater accuracy, as the nonlinear

problem is a much more difficult problem. The model was trained for 1000 iterations. The results are shown in the figure below:



(a) Solution at $t = 0$

(b) Solution at $t = 3$

(c) Solution at $t = 6$

(d) Solution at $t = 9$

Figure 2: Neural network-predicted solution for nonlinear wave equation

Unlike in the case of the linear wave equation, the predicted solution for the nonlinear case has no analytical solution to compare against. However, after comparisons with the same problem solved by a finite-difference scheme using the VisualPDE web solver, the solution appears to be of mixed reliability. While the essential features of left- and right-traveling waves are apparent, and the initial condition is met, the neural network does not appear to have evolved the solution correctly in time. However, this is mere conjecture without further, more rigorous testing.

Lastly, it is important to consider the time complexity of the algorithm; after all, meshfree methods often incur a greater computational cost in exchange for otherwise better performance. Thus, the model's training time was evaluated by training the model to 600 epochs for 2000, 5000, 7000, 10000, 15000, and 20000 domain samples. All other parameters were kept identical in the process. The results are shown in the log-log plot below:

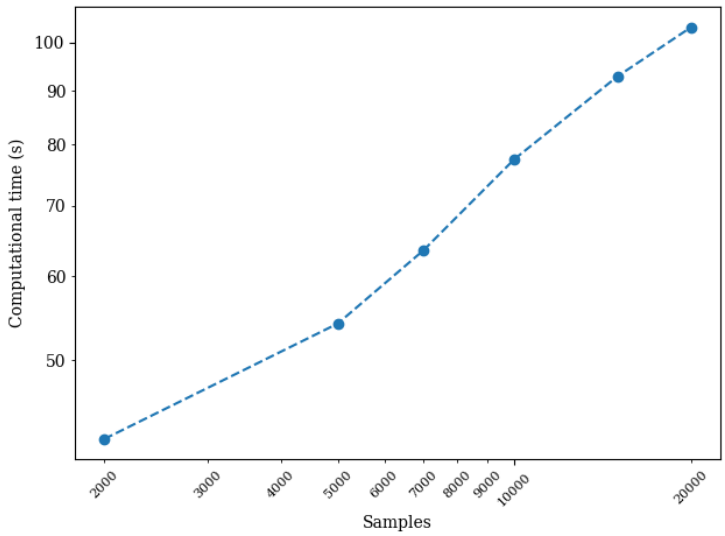


Figure 3: Training time for 2,000-20,000 samples

Although the results are indeed limited due to the relatively few test cases, one may recognize that the computational complexity of the model training is approximately $\mathcal{O}(kn)$, that is, linear in nature, rather than $\mathcal{O}(n^3)$ or worse of finite element methods[3]. This makes PINNs ideal for solving problems that require high-resolution results, such as problems in fluid mechanics, for which traditional finite elements would be prohibitively computationally-expensive.

4 Related work

The educational example outlined within this paper is a very basic demonstration of PINNs, using a rudimentary MLP as the choice of model and solving over a rectangular domain. But one may generalize PINNs to more complex domains and irregular boundary conditions by training PINNs on point clouds[5]. More sophisticated models are available, such as convolutional neural networks (CNNs)[12], which may be able to handle a larger class of problems compared to MLPs. Lastly, research is ongoing to develop neural operator models, which directly learn mappings between functional spaces to be able to use the same model to solve families of initial- and boundary-value problems [7].

5 Concluding remarks

Using neural networks for solving PDEs offer a new approach to the decades-long, well-studied field of numerical methods for PDEs. While current research has shown promising results, there are still major hurdles before PINNs see

widespread use, especially in industrial applications. The inherent difficulties in achieving satisfactory convergence in the optimization process requires further work to address. But even if they are resigned to experimental uses at the moment, PINNs show potential, and it is hoped that this paper will help facilitate a better understanding of their use to a broader audience.

6 Acknowledgements

The author would like to thank Dr. Yuri Lvov of Rensselaer Polytechnic Institute for providing the impetus to write this paper, as well as valuable feedback and mentorship. The author would also like to thank the SISA MathLab team for developing the `PINA` Python library used in the demonstration code.

7 Data availability and sharing

This research was conducted as part of Project Elara, and as such, is dedicated to the public domain. The source code, as well as other educational resources, are available at `https://github.com/elaraproject/elara-pdes-tutorial`

References

[1] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: `1502.05767 [cs.SC]`. URL: `https://arxiv.org/abs/1502.05767`.

[2] Salvatore Cuomo et al. "Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next". en. In: *J. Sci. Comput.* 92.3 (Sept. 2022).

[3] Ihor Farmaga et al. "Evaluation of computational complexity of finite element analysis". In: *2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*. 2011, pp. 213–214.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/0893-6080(89)90020-8`. URL: `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

[5] Rakhoon Hwang et al. "Solving PDEs on point clouds by physics-informed learning with graph neural networks". 2023.

[6] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980`.

[7] Nikola Kovachki et al. “Neural operator: learning maps between function spaces with applications to PDEs”. In: *The Journal of Machine Learning Research* 24.1 (Jan. 2023). ISSN: 1532-4435.

[8] Fionn Murtagh. “Multilayer perceptrons for classification and regression”. In: *Neurocomputing* 2.5 (1991), pp. 183–197. ISSN: 0925-2312. DOI: https://doi.org/10.1016/0925-2312(91)90023-5. URL: https://www.sciencedirect.com/science/article/pii/0925231291900235.

[9] P Niraula, Y Han, and J Wang. “Comparison of meshfree and mesh-based methods for boundary value problems in physics”. In: *J. Phys. Conf. Ser.* 640 (Sept. 2015), p. 012067.

[10] M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. en. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536.

[12] Zhao Zhang. “A physics-informed deep convolutional neural network for simulating and predicting transient Darcy flows in heterogeneous reservoirs without labeled data”. In: *Journal of Petroleum Science and Engineering* 211 (2022), p. 110179. ISSN: 0920-4105. DOI: https://doi.org/10.1016/j.petrol.2022.110179. URL: https://www.sciencedirect.com/science/article/pii/S0920410522000705.