

Physics-Informed Neural Networks (PINNs)

Winter Semester 2023/24 and Summer Semester 2024

[Mana Mohajer, Ph.D.](#)

Recommended References

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”, *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

S. Kollmannsberger, D. D’Angella, M. Jokeit, L. Herrmann, *Deep Learning in Computational Mechanics*, vol. 977, 978-3-030-76587-3, Springer International Publishing, 2021.

The equations and pictures in the next slides are numbered compatible with the fifth chapter of the second reference.

PINNs - Motivation

- The general motivation to develop physics-informed neural networks (PINNs): solving problems where only limited data are available, e.g. noisy measurements from an experiment.
- To compensate for the data scarcity, the algorithm is enriched with physical laws governing the problem at hand. Typically, those laws are described by parameterized non-linear partial differential equations.

PINNs - Overview

A different approach to exploit the predictive power of machine learning algorithms is to use them as surrogate models for physical simulations.

Surrogate models:

- commonly used to reduce the computational complexity of solving difficult optimization problems.
- statistical models in which the solutions are approximated by using the input-output data taken from simulations.

Generating an accurate surrogate model of a complex physical system usually requires a large amount of solution data about the problem at hand. However, data acquisition from simulations (and experiments) is often infeasible or too costly.

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”, *Journal of Computational Physics*, vol. 378, pp. 686–707, 2021.

PINNs - Overview

To overcome this limitation:

- The idea of adding prior knowledge to a machine learning algorithm has been further extended by Raissi et al. through proposing an approach to augment surrogate models with existing knowledge about the underlying physics of a problem.
- This approach is to enrich an artificial neural network with physical laws, and this new type of neural network is called Physics-Informed Neural Network (PINN).

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”, *Journal of Computational Physics*, vol. 378, pp. 686–707, 2021.

<https://github.com/ManaMohajer>

PINNs - Overview

Typically, the laws governing the physical problems are described by parameterized non-linear partial differential equations of the form:

the hidden solution $u(t, x)$, depends on time $t \in [0, T]$ and a spatial variable x

$$\frac{\partial u}{\partial t} + \mathcal{N}[u; \lambda] = 0, \quad x \in \Omega, \quad t \in \mathcal{T}. \quad (5.1)$$

\mathcal{N} represents a non-linear **differential operator** with coefficients λ

refers to a space in \mathbb{R}^D

PINNs - Overview

For stationary or transient problems, the physics-informed neural network **computes the underlying partial differential equation**.

Therefore, a classical feed-forward neural network approximates the solution $u(t, x)$ and the physics-enriched part evaluates the corresponding partial derivatives forming the left-hand side of Eq.(5.1):

$$\frac{\partial u}{\partial t} + \mathcal{N}[u; \lambda] = 0, \quad x \in \Omega, \quad t \in \mathcal{T}. \quad (5.1)$$

The network approximating $u(t, x)$, as well as the whole physics-informed neural network, both, depend on the same set of parameters. Those shared weights and biases are trained by minimizing a cost function which consists of multiple mean squared error losses.

PINNs - Overview

- The cost function in its general form can be written as

$$C = MSE_u + MSE_f \quad (5.2)$$

- MSE_u : computes the error of the approximation of $u(t, x)$ **at known data points - collocation points**
- MSE_f : enforces the partial differential equation on **a large set of randomly chosen collocation points inside the domain**

PINNs - Overview

Generally, there are two ways to enforce boundary conditions of a forward problem. The regular and constant boundary conditions can be enforced in a **strong** form. To this end, the output of the network is adapted so that the **predicted solution automatically satisfies the minimization of loss function for any given input of the domain.**

In case of a **weak** enforcement, MSE_u , Eq.(5.2) entails multiple terms that **enforce the solution at the boundary.** Note that in Mathematics, dealing with solving PDEs numerically, we use weak form by implementing weighted residual methods to solve BVPs. Based on Functional Analysis it is proven that such a solution is the best approximated solution).

The proposed approach, PINN, is not guaranteed to converge to a global minimum, and thus, an accurate solution $u(x)$, Raissi et al. showed empirically that their method achieves accurate results for different problems and architectures.

S. Kollmannsberger, D. D'Angella, M. Jokeit, L. Herrmann, *Deep Learning in Computational Mechanics*, vol. 977, 978-3-030-76587-3, Springer International Publishing, 2021.

Maziar Raissi. *maziarraissi/PINNs*. original-date: 2018-01-21T04:04:32Z. July 25, 2020. URL: <https://github.com/maziarraissi/PINNs> (visited on 07/27/2020).

<https://github.com/ManaMohajer>

Data-Driven Inference

In general, the problem of inference can be phrased as: find the hidden solution $u(t, x)$ for given coefficients λ .

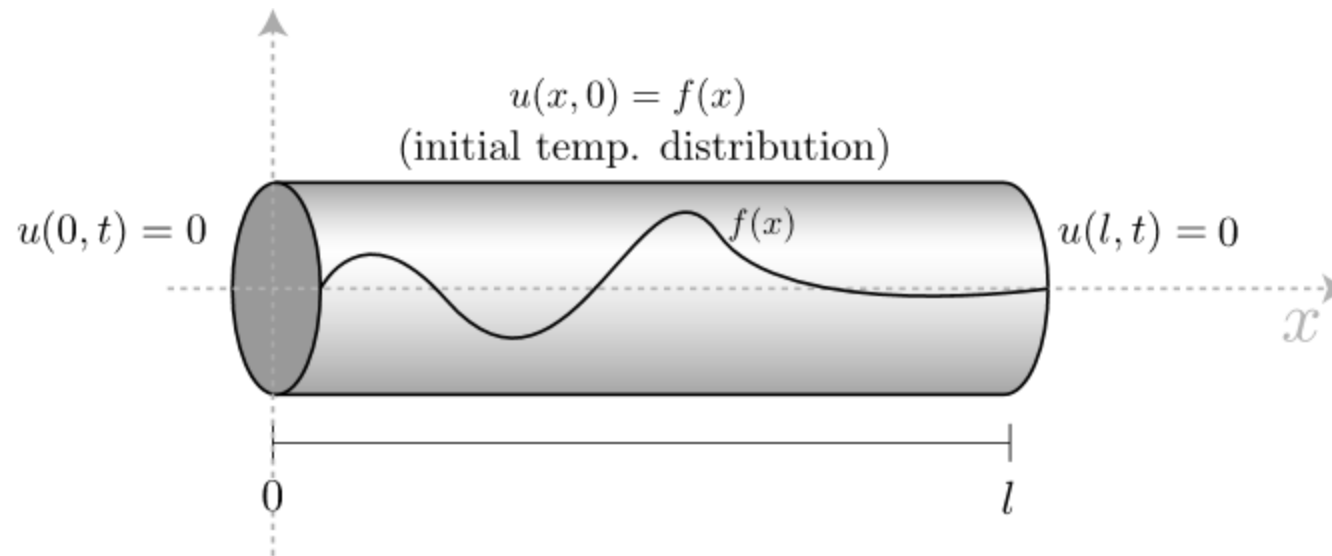
Since all values of λ are known, Eq.(5.1), simplifies to Eq. (5.3):

$$\frac{\partial u}{\partial t} + \mathcal{N}[u; \lambda] = 0, x \in \Omega, t \in \mathcal{T}. \quad (5.1)$$

$$\frac{\partial u}{\partial t} + \mathcal{N}[u] = 0, x \in \Omega, t \in \mathcal{T}. \quad (5.3)$$

Static Model

A very simple application of the physics-informed neural network is the one-dimensional linear elastic static bar/rod, which is governed by an ordinary differential equation (1D wave equation) and the corresponding boundary conditions.



Picture taken from https://upload.wikimedia.org/wikipedia/commons/9/97/Temp_Rod_homobc.svg

<https://github.com/ManaMohajer>

Static Model

The system can thereby be expressed as:

$$\underbrace{\frac{d}{dx} \left(EA \frac{du}{dx} \right)}_{\mathcal{N}[u]} + P = 0 \quad \text{on } \Omega, \quad (5.4)$$

denotes a distributed load

$$EA \frac{du}{dx} = F \quad \text{on } \Gamma_N, \quad (5.5)$$

denotes a concentrated load on Γ_N

Young's modulus E and cross-sectional area A

$$u = g \quad \text{on } \Gamma_D, \quad (5.6)$$

prescribes a displacement on Γ_D

Neumann and the Dirichlet boundaries

Static Model

In order to verify that every input fulfills this condition, the network is extended to compute the left-hand-side of Eq. (5.4)

$$f := \frac{d}{dx} \left(EA \frac{du}{dx} \right) + P \quad (5.7)$$

Since a neural network is fully differentiable, it is not only possible to compute the **derivatives with respect to the parameters necessary for training**. Likewise, the *automatic differentiation* capabilities of libraries such as PyTorch or Tensorflow allow the fast computation of **derivatives with respect to the input variable x** .

Static Model

All together, this extended network architecture can be interpreted as a physics-informed neural network as depicted in Fig. 5.1.

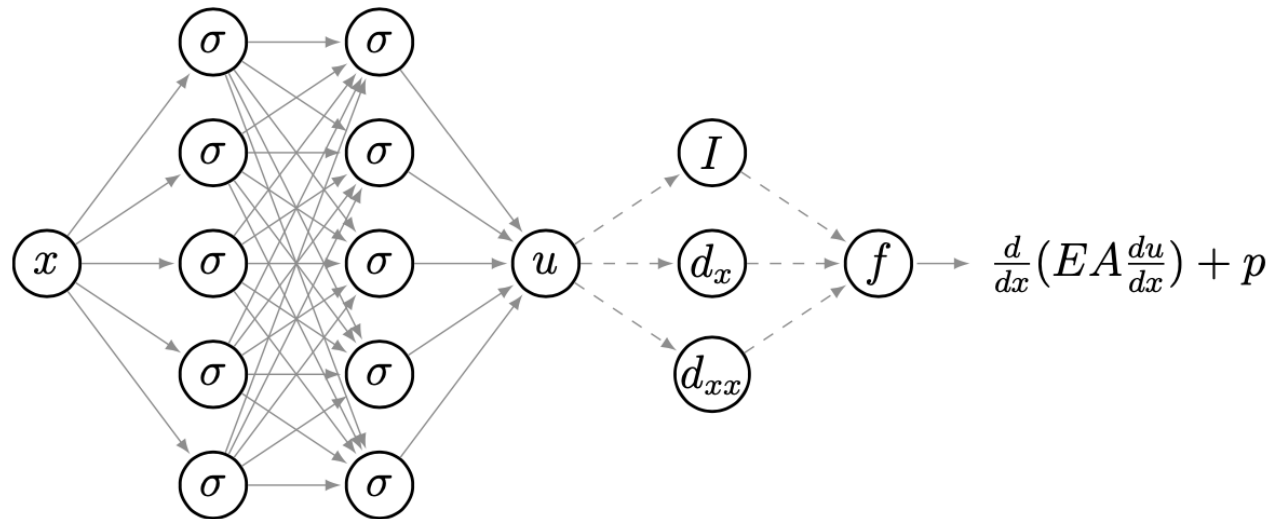


Fig. 5.1 Conceptual physics-informed neural network for the static bar equation. The left part shows the feed-forward neural network and the right part represents the physics-informed neural network. **The dashed lines denote non-trainable weights.**

Static Model

Considering the problem of linear elastic rod, we define a domain $\Omega = [0, 1]$ with $\Gamma_D = \{x \mid x = 0, x = 1\}$ and $\Gamma_N = \emptyset$. The material and geometrical parameters are set to $EA = 1$. A solution for the displacement is chosen as:

$$u(x) = \sin(2\pi x), \quad (5.8)$$

After inserting into the differential equation (5.4), it results in the following distributed load:

$$p(x) = 4\pi^2 \sin(2\pi x), \quad (5.9)$$

Furthermore, the following Dirichlet boundary conditions apply:

$$u(0) = u(1) = 0 \quad (5.10)$$

Static Model

With this information, it is now possible to define a cost function. In case of the static physics-informed neural network, this custom cost function is assembled from two mean squared error losses (Eq. (5.2)):

$$C = MSE_b + MSE_f \quad (5.11)$$

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} (u_{NN}(x_b^i) - u_b^i)^2, \quad (5.12)$$

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} (f_{NN}(x_f^i))^2, \quad (5.13)$$

Static Model

The physics-informed network is implemented in PyTorch.

At first, a neural network is created to predict the displacement, $u(x)$.

For the sake of simplicity, a model consisting of a single hidden layer with the dimension of `hidden_dim` is defined.

The input and output dimensions are correspondingly given as `input_dim` and `output_dim`. The linear transformations from one layer to the next are defined by `torch.nn.Linear` and the non-linear activation function is chosen as the hyperbolic tangent, `torch.nn.Tanh`. There is no activation function in the output layer meaning that the range of the outputs is not limited.

Static Model

Before opening the Jupyter notebook and going through the code for this example, we go through some blocks of the code and some descriptions.

```
def buildModel(input_dim, hidden_dim, output_dim):  
    model = torch.nn.Sequential(torch.nn.Linear(input_dim, hidden_dim),  
                                torch.nn.Tanh(),  
                                torch.nn.Linear(hidden_dim, output_dim))  
    return model
```

Static Model

With the neural network, **the first part of the PINN** illustrated in Fig. 5.1 is defined. A displacement prediction, \hat{u} , at $x = 0.5$ can be calculated with the following code.

```
model = buildModel(1, 10, 1)
u_pred = model(torch.tensor([0.5]))
```

The **second part of the network in Fig. 5.1.** requires the computation of the derivatives with respect to the input, x . This is achieved with **PyTorch's built-in automatic differentiation**, which creates and retains a computational graph to store all information necessary for calculating the gradients.

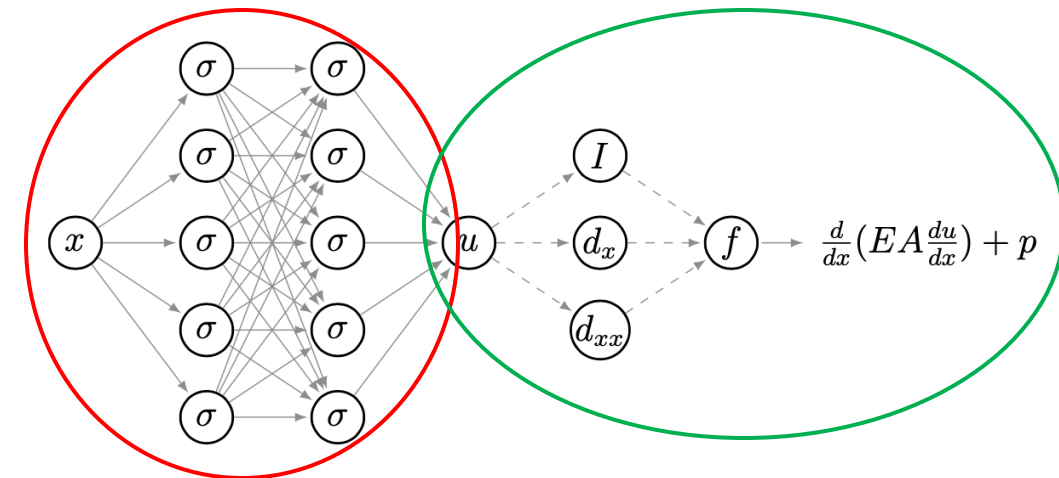


Fig. 5.1

Static Model

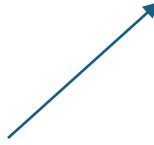
Finally, f , as in Eq. (5.7),

$$f := \frac{d}{dx} \left(EA \frac{du}{dx} \right) + P \quad (5.7)$$

is computed with the following code.

```
def f(model, x, EA, p):  
    u = model(x)  
    u_x = get_derivative(u, x)  
    EAU_xx = get_derivative(EA(x) * u_x, x)  
    f = EAU_xx + p(x)  
    return f
```

Be careful while going through code!



Static Model

Now, to calculate the loss function of the differential equation MSE_f , the example values are inserted into the model.

```
model = buildModel(1, 10, 1)
x = torch.linspace(0, 1, 10, requires_grad=True).view(-1, 1)
EA = lambda x: 1 + 0 * x
p = lambda x: 4 * math.pi**2 * torch.sin(2 * math.pi * x)

f = f(model, x, EA, p)
MSE_f = torch.sum(f**2)
```

Static Model

In combination with the loss function of the boundary conditions MSE_b , allowing to compute the cost function, C , from Eq. (5.11). The implementation for this example is shown in the code snippet below.

```
model = buildModel(1, 10, 1)
u0 = 0
u1 = 0

u0_pred = model(torch.tensor([0.]))
u1_pred = model(torch.tensor([1.]))
MSE_b = (u0_pred - u0)**2 + (u1_pred - u1)**2
```

Static Model

It is now possible to compute and minimize the cost function C via a stochastic gradient approach or more sophisticated methods such as the L-BFGS optimizer.

When the cost function is sufficiently small, an accurate prediction of the displacement $u(x)$ is to be expected.

In principle, this is how the physics-informed neural network works. The entire training procedure is summarized in Algorithm 4 in the next slide.

Static Model

Algorithm 4 Training a physics-informed neural network for the static solution of the problem described in Eq. (5.4).

Require: training data for boundary condition $\{x_b^i, u_b^i\}_{i=1}^{N_b}$

generate N_f collocation points with a uniform distribution $\{x_f^i\}_{i=1}^{N_f}$

define network architecture (input, output, hidden layers, hidden neurons)

initialize network parameters Θ : weights $\{W^l\}_{l=1}^L$ and biases $\{b^l\}_{l=1}^L$ for all layers L

set hyperparameters for L-BFGS optimizer (*epochs*, learning rate α , ...)

for all *epochs* **do**

$\hat{u}_b \leftarrow u_{NN}(x_b; \Theta)$

$f \leftarrow f_{NN}(x_f; \Theta)$

 compute MSE_b, MSE_f

 ▷ cf. Eqs. (5.12) and (5.13)

 compute cost function: $C \leftarrow MSE_b + MSE_f$

 update parameters: $\Theta \leftarrow \Theta - \alpha \frac{\partial C}{\partial \Theta}$

 ▷ L-BFGS

end for

Static Model

- Limited-memory BFGS (L-BFGS or LM-BFGS) is an optimization algorithm, in the family of quasi-Newton methods
- It approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) using a limited amount of computer memory
- A popular algorithm for parameter estimation in machine learning, with the algorithm's target problem of minimizing a function, $f(x)$, over unconstrained values of the real-vector x where f is a differentiable scalar function.

Liu, D. C.; Nocedal, J. (1989). "On the Limited Memory Method for Large Scale Optimization". *Mathematical Programming B*. 45 (3): 503–528.

Malouf, Robert (2002). "A comparison of algorithms for maximum entropy parameter estimation". *Proceedings of the Sixth Conference on Natural Language Learning (CoNLL-2002)*. pp. 49–55.

Andrew, Galen; Gao, Jianfeng (2007). "Scalable training of L_1 -regularized log-linear models". *Proceedings of the 24th International Conference on Machine Learning*.

<https://github.com/ManaMohajer>

Parameters of L-BFGS Class

CLASS torch.optim.LBFGS (params, lr=1, max_iter=20, max_eval=None, tolerance_grad=1e-07, tolerance_change=1e-09, history_size=100, line_search_fn=None)

Parameters	
lr (float)	learning rate (default: 1).
max_iter (int)	maximal number of iterations per optimization step (default: 20).
max_eval (int)	maximal number of function evaluations per optimization step (default: max_iter * 1.25).
tolerance_grad (float)	termination tolerance on first order optimality (default: 1e-5).
tolerance_change (float)	termination tolerance on function value/parameter changes (default: 1e-9).
history_size (int)	update history size (default: 100).
line_search_fn (str)	either 'strong_wolfe' or None (default: None).

Table 1. L-BFGS class arguments (<https://pytorch.org/docs/stable/generated/torch.optim.LBFGS.html>)

Static Model

- Difference in the order of magnitudes between the cost function of the differential equation and the cost function of the boundary conditions.
- Challenge: Adjusting this difference with a weighting factor on the boundary condition due to the difficulty of determining a priori.
- An alternative is to enforce the Dirichlet boundary conditions strongly, leading to an unconstrained optimization problem: only the differential equation cost has to be minimized.
- Now, open the Jupyter notebook and go through the code for this example. Afterwards we continue with the rest of slides!