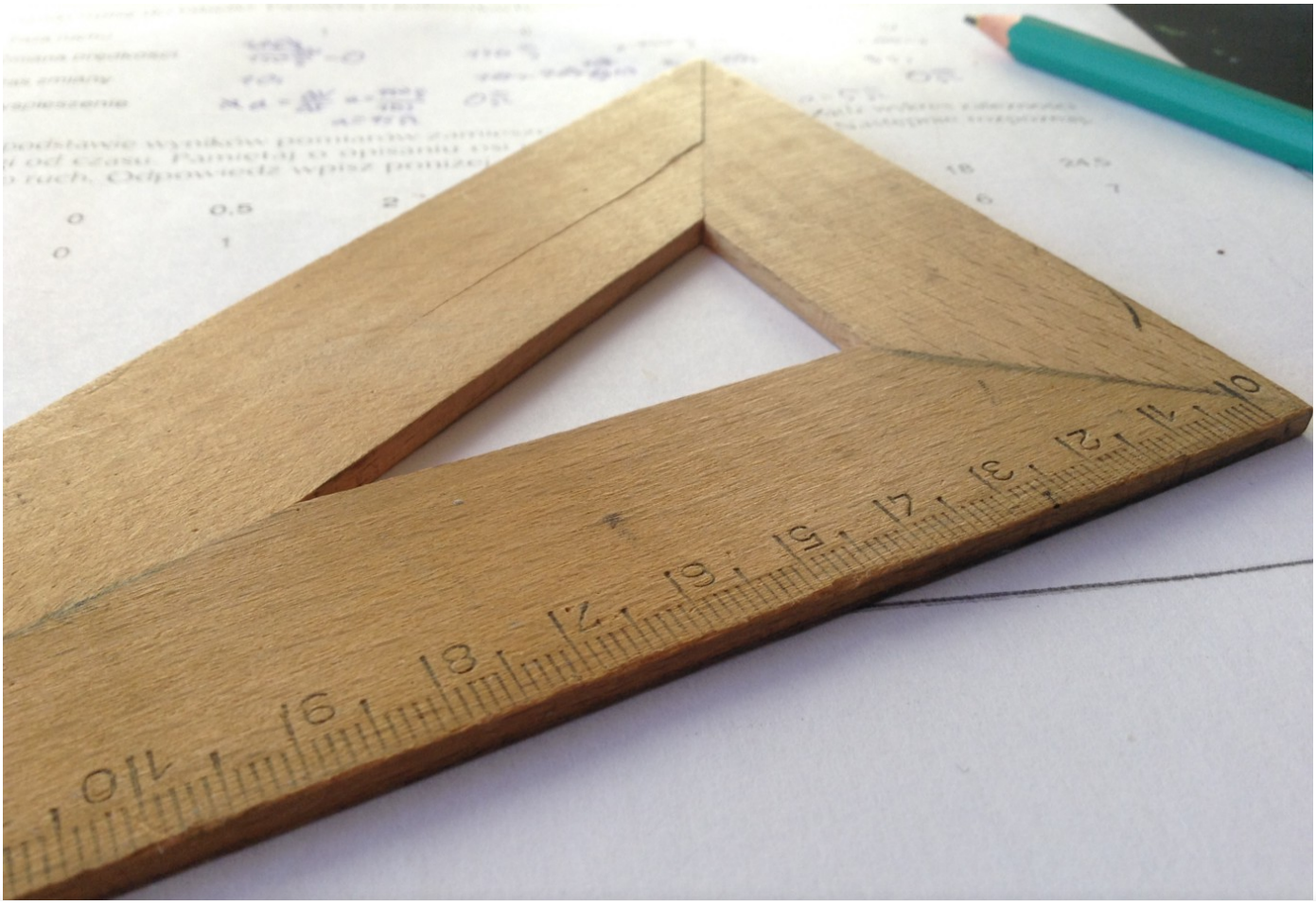
Published in Towards Data Science

Mario Dagrada Follow

May 1 · 8 min read · Listen

Solving Differential Equations with Neural Networks

A hands-on introduction to physics-informed neural networks with PyTorch



Over the last decades, artificial neural networks have been used to solve problems in varied applied domains such as computer vision, natural language processing and many more. Recently, another very promising application has emerged in the scientific machine learning (ML) community: The solution of partial differential equations (PDEs) using artificial neural networks, using an approach normally referred to as **physics-informed neural networks** (PINNs). Today, PINNs are not limited anymore to a pure research topic but are gaining traction in industry as well, enough to enter the famous Gartner hype cycle for emerging technologies in 2021.

PDEs play a crucial role in many fields of engineering and fundamental science ranging from fluid dynamics to acoustic and structural engineering. Finite elements modeling (FEM) methods are the standard solvers employed ubiquitously in the industry. Despite their popularity, FEM methods display some limitations such as their computational cost for large industrial problems (mainly due to the required mesh size) and issues in leveraging external data sources, such as sensors data, to drive the solution of the PDEs.

The PINNs approach discussed in this post is regarded as a promising alternative to FEM methods for covering some of these limitations. This approach is quite different from the standard supervised ML. In fact, instead of relying purely on data, it uses the physical properties of the PDE itself to guide the training process. Known data points can be easily added on top of the physics-based loss function to speed-up training speed.

This post gives a simple introduction to the main concepts behind PINNs and then shows how to build a PINN from scratch to solve a simple first-order ordinary differential equation. For building the neural network, I will use the amazing PyTorch library. Let's get started!

How physics-informed NNs work

To gain a high-level understanding of PINNs, let's begin with choosing a differential equation. To keep things simple, in this post I focus on the logistic equation, a famous model of population growth developed in the 19th century:

Here the function *f(t)* represents the population growth rate over time *t* and the parameter *R* yields the maximum population growth rate. In order to fully specify the solution of this equation, one needs to impose a boundary condition, for example at $t = 0$ such as:

$$f(t=0) = 1$$

Boundary condition for the logistic equation.

Even though the solution to this equation is known analytically, it represents a simple playground to illustrate how PINNs work. All the techniques explained in the following are readily applicable to more complex ordinary and partial differential equations.

PINNs are based on two fundamental properties of NNs:

- It has been formally demonstrated [1] that NNs are **universal function approximators**. Therefore a NN, provided that it is deep and expressive enough, can approximate any function and therefore also the solution for the logistic equation above.
- It is easy and cheap to compute the derivatives (of any order) of a NN output with respect to any of its input (and of course model parameters during backpropagation) using **automatic differentiation** (AD). AD is actually what made neural networks so efficient and successful in the first place.

These are nice features but how can we make the NN actually learn the solution? And here it comes the surprisingly simple but extremely clever idea behind PINNs [2, 3]: We can construct the NN loss function such that, when minimized, the PDE is automatically satisfied. In other words, the most important loss contribution is taken as the residual of the differential equation as follows:

$$\frac{\mathrm{d}f_{NN}(t)}{dt} - Rt(1-t) = 0$$

where f_*NN(t)* is the output of a NN with one input and its derivative is computed using AD. It is immediate to see that if the NN output respects the equation above, one is actually solving the logistic equation. To compute the actual loss contribution coming from the DE residual, one needs to specify a set of points in the equation domain (usually referred to as **colocation points**) and evaluate the mean square error (MSE) or another loss function as an average over all the chosen coordinates:

$$\mathcal{L}_{DE} = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{\mathrm{d}f_{NN}}{\mathrm{d}t}\bigg|_{x_j} - R\,x_j(1-x_j)\right)^2$$

Loss contribution given by the differential equation residual averaged over a set of collocation points.

However, a loss based only on the above residual does not ensure to have a unique solution to the equation. Therefore, let's include the boundary condition by adding it to the loss computation in exactly the same way as above:

$$\mathcal{L}_{BC} = \left(f_{NN}(t_0) - 1\right)^2 \quad \text{with } t_0 = 0$$

Boundary conditions loss contribution added to the MSE loss.

Hence, the final loss simply reads:

$$\mathcal{L} = \mathcal{L}_{DE} + \mathcal{L}_{BC}$$

During the optimization procedure, this is minimized and the NN output is trained to respect both the differential equation and the given boundary condition, thus approximating the final DE solution.

The PINN framework is very flexible and, using the ideas presented above, one can add more boundary conditions, include more complex ones such as constraints on the derivatives of *f(x)*, or treat time-dependent and multidimensional problems using a NN

Build a PINN from scratch

The neural network

The main ingredient of a PINN is of course the neural network itself. For this post, we choose a basic NN architecture composed by a stack of linear layers with standard *tanh* activation functions. Since we have one independent variables, the time t, the NN should take one feature as input and return one output which represents the best DE solution guess given the current model parameters. Below a PyTorch implementation of this architecture where the number of neurons and hidden layers are left as input (hyper)parameters.

```
from torch import nn

class NNApproximator(nn.Module):
    """Simple neural network accepting one feature as input and returning a single output

    In the context of PINNs, the neural network is used as universal function approximator
    to approximate the solution of the differential equation
    """
    def __init__(self, num_hidden: int, dim_hidden: int, act=nn.Tanh()):
        super().__init__()

        self.layer_in = nn.Linear(1, dim_hidden)
        self.layer_out = nn.Linear(dim_hidden, 1)

        num_middle = num_hidden - 1
        self.middle_layers = nn.ModuleList(
            [nn.Linear(dim_hidden, dim_hidden) for _ in range(num_middle)]
        )
        self.act = act

    def forward(self, x):
        out = self.act(self.layer_in(x))
        for layer in self.middle_layers:
            out = self.act(layer(out))
        return self.layer_out(out)
```

PINNs are a very active research area and much more complex and often problem-tailored neural network architectures have been devised. Discussion on these architectures is outside the scope of this introductory blog.

Build the loss function

Now that we defined our universal function approximator, let's build the loss function. As discussed, this is composed by the DE residual term, that acts as a physics-informed regularization, and the boundary condition term, driving the network to converge to the desired solution among the infinite possible ones.

First of all, one needs to choose a set of colocation points. Since we are solving a very simple problem, we can choose a uniformly spaced grid in the time domain: `t = torch.linspace(0, 1, steps=10, requires_grad=True)` . For more complex problems, the choice of colocation points is very important and requires a much more careful choice.

The DE residual loss needs the evaluation of the derivatives of the NN output with respect to its input:

```
import torch

def f(nn: NNApproximator, x: torch.Tensor) -> torch.Tensor:
    """Compute the value of the approximate solution from the NN model"""
    return nn(x)


def df(nn: NNApproximator, x: torch.Tensor = None, order: int = 1) -> torch.Tensor:
    """Compute neural network derivative with respect to the input feature(s) using PyTorc
    df_value = f(nn, x)
    for _ in range(order):
        df_value = torch.autograd.grad(
            df_value,
            x,
            grad_outputs=torch.ones_like(x),
            create_graph=True,
            retain_graph=True,
```

Compute solution guess and its derivatives using PyTorch autograd engine.

The code above uses the PyTorch `autograd` engine to compute the derivatives with respect to time *t* automatically. Even though the logistic equation only needs first-order derivatives, the code shows also that by a repeated application of the `torch.autograd.grad` function, arbitrary order derivatives can be computed. This is equivalent to perform multiple backward passes. Using the functions above, the MSE loss is easily computed as a sum of the DE contribution at each colocation point and the boundary contribution:

```
T0 = 0.0 # initial time
F0 = 1.0 # boundary condition value

# DE contribution
interior_loss = df(nn, x) - R * x * (1 - x)

# boundary contribution
boundary = torch.Tensor([T0])
boundary.requires_grad = True
boundary_loss = f(nn, boundary) - F0

final_loss = \
  # average over all the colocation points
  interior_loss.pow(2).mean() + \
  # boundary contribution is just a single value
  boundary_loss ** 2
```

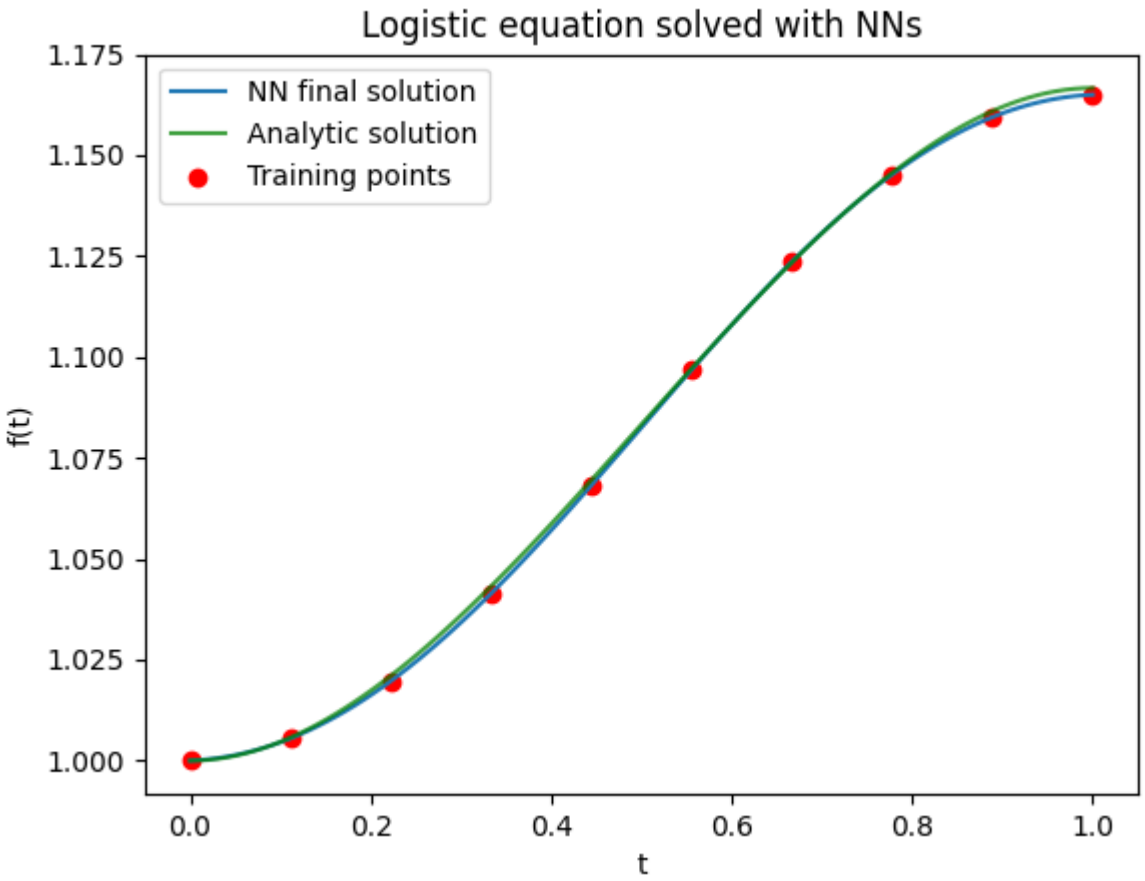
pinn_loss.py hosted with ❤ by GitHub view raw

That's it! The custom loss define above ensures that after the training procedure, the NN will approximate the solution to the logistic equation. Now, let's see it in action.

Solving the logistic equation with PINNs

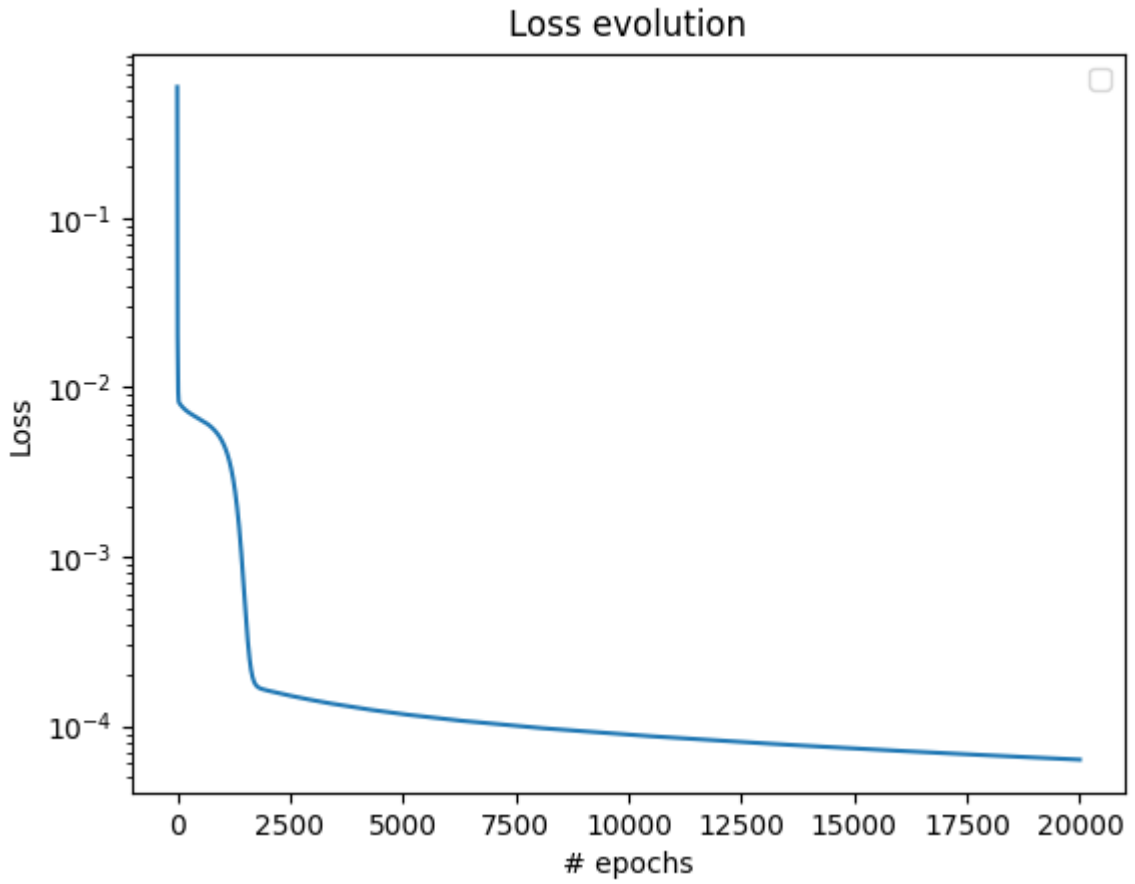
Since the loss defined above is built using only differentiable functions, we can directly compute the gradients with respect to the model parameters using the backward pass (one line of code in PyTorch): `final_loss.backward()`. The optimization procedure is then

Let's see some results. We use the stochastic gradient descent optimizer with a learning rate of 0.1 with just 10 training points. Given the simplicity of the logistic equation, 20000 epochs are enough to almost perfectly reproduce the analytical result with maximum growth rate set at $R = 1$:



Source: Image by Author.

In the plot above, the solution is evaluated on 100 uniformly spaced points and I show also the 10 training points for comparison. The loss landscape looks like this:



Source: Image by Author.

Notice that using the Adam optimizer, an even better result can be obtained with a much lower number of epochs. However, optimizing the result is out of the scope of this post which only aims at explaining how PINNs work.

Here we solved a very simple, one dimensional problem. With more complex equations, convergence is not achieved so easily. Particularly for time-dependent problems, many useful tricks have been devised over the past years such as decomposing the solution domain in different parts solved using different neural networks, smart weighting of different loss contributions to avoid converging to trivial solutions and many more. I will introduce some of these tricks in future posts, so stay tuned.

further advances are needed to make PINNs routinely applicable to industrial problems, they are a really active and exciting area of research and represent a promising alternative to standard differential equation solvers.

The complete code of this post can be found on GitHub. Do not hesitate to contact me on LinkedIn for any question or remark on the post.

References and further readings

[1] Kurt Hornik, Maxwell Stinchcombe and Halbert White, *Multilayer feedforward networks are universal approximators,* Neural Networks **2**, 359–366 (1989)

[2] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang and Liu Yang, *Physics informed machine learning*, Nature Reviews Physics **3**, 422–440 (2021)

[3] Ben Mosley, So, what is a physics-informed neural network? — Ben Moseley