<https://towardsdatascience.com/solving-differential-equations-with-neural-networks-afdcf7b8bcc4>

<https://github.com/madagra/basic-pinn>

# Introduction to Physics-informed Neural Networks

## A hands-on tutorial 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, with an approach normally referred to as **physics-informed neural networks** (PINNs). PINNs have been originally introduced in the seminal work in [1] and today they 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 we focus on the [logistic differential equation](https://mathworld.wolfram.com/LogisticEquation.html), a famous first order ordinary differential equation used to model population growth:

A black background with a black square

Description automatically generated with medium confidence

First order logistic differential equation for modeling population growth

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



Boundary condition for the logistic equation.

Even though the solution to this equation can be easily derived 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. However, for more complex scenarios additional tricks will be needed in order to get good convergence.

PINNs are based on two fundamental properties of NNs:

* It has been formally demonstrated [2] 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 differential 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 [3, 4]: 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:

A black background with a black square

Description automatically generated with medium confidence

Differential equation residual.

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 differential 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 colocation points:

A black background with a black square

Description automatically generated with medium confidence

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:



Boundary conditions loss contribution added to the MSE loss.

Hence, the final loss simply reads:



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 with multiple inputs.

Let’s now see how to construct such loss function with a simple neural network built with PyTorch.

# 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.

class LinearNN(nn.Module):  
 def \_\_init\_\_(  
 self,  
 num\_inputs: int = 1,  
 num\_layers: int = 1,  
 num\_neurons: int = 5,  
 act: nn.Module = nn.Tanh(),  
 ) -> None:  
 """Basic neural network architecture with linear layers  
   
 Args:  
 num\_inputs (int, optional): the dimensionality of the input tensor  
 num\_layers (int, optional): the number of hidden layers  
 num\_neurons (int, optional): the number of neurons for each hidden layer  
 act (nn.Module, optional): the non-linear activation function to use for stitching  
 linear layers togeter  
 """  
 super().\_\_init\_\_()  
   
 self.num\_inputs = num\_inputs  
 self.num\_neurons = num\_neurons  
 self.num\_layers = num\_layers  
  
 layers = []  
  
 # input layer  
 layers.append(nn.Linear(self.num\_inputs, num\_neurons))  
  
 # hidden layers with linear layer and activation  
 for \_ in range(num\_layers):  
 layers.extend([nn.Linear(num\_neurons, num\_neurons), act])  
  
 # output layer  
 layers.append(nn.Linear(num\_neurons, 1))  
  
 # build the network  
 self.network = nn.Sequential(\*layers)  
  
 def forward(self, x: torch.Tensor) -> torch.Tensor:  
 return self.network(x.reshape(-1, 1)).squeeze()

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=30, requires\_grad=True) or we can randomly sample new colocation points from the time domain at each iteration of the optimizer. For more complex problems, the choice of colocation points is extremely important and can strongly affect the results.

For computing the model output and its derivative we are using the functional API of PyTorch which makes models fully functional by decoupling the parameters from the model itself. You can check my [other post](https://towardsdatascience.com/introduction-to-functional-pytorch-b5bf739e1e6e) on the subject for more details.

The functional approach to PyTorch is very convenient when dealing with (higher-order) derivatives of the NN output with respect to its inputs, as often the case for PINNs. In the code below, we use the torch.func API introduced in PyTorch 2.0 to build a functional forward pass and higher-order gradient calculations with support for batches

from torch.func import functional\_call, grad, vmap  
  
model = LinearNN()  
  
# notice that `functional\_call` supports batched inputs by default  
# thus there is not need to call vmap on it, as it's instead the case  
# for the derivative calls  
def f(x: torch.Tensor, params: dict[str, torch.nn.Parameter]) -> torch.Tensor:  
 return functional\_call(model, params\_dict, (x, ))  
  
# return function for computing higher order gradients with respect  
# to input by simply composing `grad` calls and use again `vmap` for  
# efficient batching of the input  
dfdx = vmap(grad(f), in\_dims=(0, None))  
d2fdx2 = vmap(grad(grad(f)), in\_dims=(0, None))

Notice that calls to grad can be composed without restrictions, thus allowing to compute derivatives of any order with respect to the inputs. Using the functions defined above, the MSE loss is easily computed as a sum of the DE contribution at each colocation point and the boundary contribution. Notice that, given the functional nature of the forward pass and derivatives, the loss function must also take the model params as input argument:

import torch  
import torch.nn as nn  
  
R = 1.0 # rate of maximum population growth parameterizing the equation  
X\_BOUNDARY = 0.0 # boundary condition coordinate  
F\_BOUNDARY = 0.5 # boundary condition value  
  
def loss\_fn(params: torch.Tensor, x: torch.Tensor):  
  
 # interior loss  
 f\_value = f(x, params)  
 interior = dfdx(x, params) - R \* f\_value \* (1 - f\_value)  
  
 # boundary loss  
 x0 = X\_BOUNDARY  
 f0 = F\_BOUNDARY  
 x\_boundary = torch.tensor([x0])  
 f\_boundary = torch.tensor([f0])  
 boundary = f(x\_boundary, params) - f\_boundary  
  
 loss = nn.MSELoss()  
 loss\_value = loss(interior, torch.zeros\_like(interior)) + loss(  
 boundary, torch.zeros\_like(boundary)  
 )  
  
 return loss\_value

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

## Solving the differential equation with PINNs

PyTorch does not currently offer native support for optimizers with the functional API we used. However, the amazing PyTorch community comes to rescue and one can get a functional version of most of PyTorch optimizers using the [torchopt library](https://github.com/metaopt/torchopt). The interface of this library is very intuitive and will be immediately familiar to any PyTorch user. A basic training loop for our functional model is shown below. Notice that we sample randomly the solution domain at each iteration.

# choose the configuration for the training loop  
batch\_size = 30 # number of colocation points to sample in the domain  
num\_iter = 100 # maximum number of iterations  
learning\_rate = 1e-1 # learning rate  
domain = (-5.0, 5.0) # ;ogistic equation domain  
  
# choose optimizer with functional API using functorch  
optimizer = torchopt.FuncOptimizer(torchopt.adam(lr=learning\_rate))  
  
# initial parameters randomly initialized  
params = tuple(model.parameters())  
  
# train the model  
for i in range(num\_iter):  
  
 # sample points in the domain randomly for each epoch  
 x = torch.FloatTensor(batch\_size).uniform\_(domain[0], domain[1])  
  
 # compute the loss with the current parameters  
 loss = loss\_fn(params, x)  
  
 # update the parameters with functional optimizer  
 params = optimizer.step(loss, params)  
  
 print(f"Iteration {i} with loss {float(loss)}")

Let’s see some results. We use the Adam optimizer with a learning rate of 0.1 with 30 training points sampled uniformly from the domain at each optimization epoch. Given the simplicity of the chosen differential equation, 100 epochs are enough to almost perfectly reproduce the analytical result with a maximum growth rate set at R = 1:

A graph with red dots and green lines

Description automatically generated

The solution of the logistic equation using the physics informed neural network approach. A set of random training points is also shown.

In the plot above, the solution is evaluated on 100 uniformly spaced points, the evolution of the loss per each epoch (where the y-axis is in logarithmic scale) looks like this:

A graph with blue lines

Description automatically generated

The evolution of the loss function values with the Adam optimizer. The plot has a logarithmic scale on the y axis.

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.

This post gives a simple, high-level introduction to physics-informed neural networks, a promising machine learning method to solve (partial) differential equations. Although 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](https://github.com/madagra/basic-pinn). Do not hesitate to contact me on [LinkedIn](https://www.linkedin.com/in/mariodagrada/) for any question or remark on the post.

# References and further readings

[1] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational physics (2019)

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

[3] 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)

[4] Ben Mosley, [So, what is a physics-informed neural network? — Ben Moseley](https://benmoseley.blog/my-research/so-what-is-a-physics-informed-neural-network/)