Project: Physics-Informed Neural Networks (PINNs)

This note is written by Allan P. Engsig-Karup (apek@dtu.dk)

This project focus on learning about physics-informed neural networks (PINNs) that can be viewed as a class of mathematical techniques that belongs to what can be viewed as an emerging set techniques within the area of scientific machine learning (SciML). SciML can be applied across all areas of science and engineering, either to extract insights from data or to transform engineering solution using combined and emerging techniques from machine learning and scientific computing. Be aware that the techniques evolve fast and can be applied in many ways to address problems of various kinds and hence there is not a single "correct" way to view these methods, which can complicate how to understand what they can do. However, one starting point is to view neural networks through the lens of mathematical function approximation, which can be done in various ways as we will explore.

Getting setup

To work with PINNs it is recommended to install one of the distributions Anaconda or Miniconda (lightweight version) to be able to work with python-based open source software on your laptop, e.g. through using Jupyter notebooks that is an emerging format for sharing scientific calculations in a standardized document. This can be done via JupyterLab as the notebook interface. If you are familiar with installing packages etc. then it is possible to just install the python packages and modules that you need individually. Also, it is possible to use Google's online Colaboratory (colab) or Collaborative Calculation and Data Science (CoCalc) that runs directly in a web browser without any additional software installations and also makes it possible to use some of the high performance computing ressources available. Also, colab comes with tutorials etc, e.g. see Machine Learning Hello World example using Tensorflow or Tensorflow examples. To work with PINNs, it is possible to find examples and use packages that builds on Google's open source tensorflow framework, e.g. using Keras with examples given in Jupyter Notebooks. Also, it is possible to use Facebook's PyTorch or Google's Jax with fast automatic differentiation for high-performance machine learning research. A main advantage of using open source frameworks is that they have been prepared for high performance computing and comes with automatic differentiation support. Examples of packages for PINN that can be used as a starting point are DeepXDE or SciANN. Useful developments are also available via the packages in Julia for SciML. It is even possible to do PINNs in Matlab with motivating examples highlighted in . In all frameworks mentioned here, there are some documentation and examples that can provide a suitable starting point, albeit with some learning curve since the developments can be viewed as recent work that is still in progress and starts to see maturity. To create reproducible scientific results it is possible to use open source technology such as Binder 2.0 to quickly create and share computational environments with collaborators needed to interact with the code, notebooks and data that is stored in a Github repository.

Using conda, it is possible to easily share your configuration and get it to work on any computer using any operating system where conda is installed. To create an environment "ml" with python version 3.7.6 installed, we can do it as follows

```
> create a new environment: conda create -n ml python=3.7.6
```

Once created, we can activate it and install all python modules / packages from a text file 'requirements.txt' that we need for our setup through the commands

```
> activate the environment: conda activate ml
> install python packages: pip install -r requirements.txt
```

Hence, always provide the file with your source code to make the setup reproducible.

If we do not need it or it we messed something up, we can delete it the 'ml' environment. To delete an environment that was created earlier, we use the commands

```
> conda env remove --name ml
```

Getting started

To start on the journey of learning about PINNs, you will need to both learn about the theoretical foundations of Neural Networks (fx. consult the Deep Learning Book) and getting aquatinted with the practical coding within the frameworks starting from examples and utilizing numerical analysis and dynamical systems modelling and theory. A key idea behind PINNs is to consider them as a general framework that can be used for solving a broad range of problems related to forward problems (simulation using mathematical models), system identification (model discovery from data) and inverse problems (parameter estimation of model parameters through using measurements/data). There are vast number of potential applications across science and engineering, if the techniques are understood well and applied in the right context. Mathematical optimization and techniques to formulate the objectives are used to pose the *learning problem* that is needed to obtain an approximate model defined in terms of Neural Networks. The neural network is often used to represent the solution to a problem defined in terms of differential equations, and suitable boundary and initial conditions. The use of mathematical models in this way is a way to make the modelling physics-informed and explainable.

In the following, we will learn that there are fundamentally two different ways to build function approximators in data-driven approaches. The first approach is to build models directly from data only, and the second approach is to build models using physics-constrained approaches where mathematical models are used in the learning process. These two approaches can also be combined into what we then called hybrid models that can then be viewed as an approach for combining physics-based modelling and measurement data.

Function Approximation using Neural Networks

As a starting point for understanding how to work with Neural Networks to generate function approximations it is suggested to work through a set of exercises to gain confidence in proper use and understanding of the techniques as well as key concepts. This is a prerequisite for developing more advanced approaches on your own, namely, master the fundamentals and try and understand pros and cons.

a) Neural Networks are universal function approximators[1, 2, 3]. Define a function $f : x \mapsto y$ and seek a function approximation using a Neural Network representation $N(x)$ such that $N(x) \approx f(x)$. To learn the function approximation it is possible to use supervised learning where data samples are provided as a set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{M}$. Try to do this in 1D and 2D. Try with various functions such as $f(x) = ax^3 + bx^2 + cx$, $f(x) = \sin(x)$, $f(x) = |x|$, $f(x) = \exp(\sin(x))$, etc. What can you conclude on how well it works to use neural networks to approximate functions of different smoothness?

b) From the neural network representation compute using automatic differentiation the derivatives of the function in 1D and 2D and compare to the analytical derivatives.

c) Try to take into account the function values $f(x)$ as well as the derivatives $f'(x)$ at different points x in the learning procedure to see if you can improve the accuracy of the learned function approximations. Does the accuracy improve? Evaluate the accuracy both inside the domain (interpolation) of the samples and outside this domain (extrapolation).

d) How much data (=number of sample pairs (x_i, y_i) used in training) is needed to generate the function approximations for a given accuracy tolerance? Try and estimate this, e.g. by plotting the number of parameters of the network vs. the accuracy achieved against exact functions.

e) Try and evaluate the learned Neural Network. How fast are the predictions when using the model for inference (evaluations using new samples)? How much time did it take to train the model in the first place? Try and evaluate the performance of a neural network by estimating the computational complexity (plot training time vs. number of parameters in the network for increasing number of parameters) of the training procedure as well as the inference (prediction) step separately.

Physics Informed Neural Networks

Physics-informed neural networks (PINNs)[4,5,6,7] are a scientific machine learning technique for solving ordinary and partial differential equation (ODE/PDE) problems without using large volumes of data through using instead the definition of the ODE/PDE problem. Hence, the objective of PINNs are to generate approximate solutions to ODEs/PDEs by training a neural

[1] G. Cybenko, “Approximation by superpositions of a sigmoidal function,”Mathematics of Control, Signals and Systems 2, no. 4 (December 1989): 303–314; A.E. Barron, “Universal approximation bounds for superpositions of a sigmoidal function,”IEEE Trans. on Information Theory 39 (June 1993): 930–945.

[2] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators” Neural Networks 2, no. 5 (1989): 359–366.

[3] N.J. Guliyev and V.E. Ismailov, “A Single Hidden Layer Feedforward Network withOnly One Neuron in the Hidden Layer Can Approximate Any Univariate Function,”NeuralComputation28, no. 7 (2016): 1289–1304.

[4] Hyuk Lee. Neural Algorithm for Solving Differential Equations, Journal of computational physics 91, 110-131, 1990.

[5] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. IEEE Tranactionos on Neural Networks, 9 (5), 987-1000, 1998.

[6] Maziar Raissi, Paris Perdikaris, George Em Karniadakis. Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. arXiv 1711.10561.

[7] Maziar Raissi, Paris Perdikaris, George Em Karniadakis. Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. arXiv 1711.10566.

network through minimizing a loss function that incorporate the ODE/PDE as well as constraints such as boundary conditions. Also initial values needs to be provided in the case the problem is an initial value problem. In the following, we will consider how to solve a boundary value problem and an initial value problem using PINNs. A good book on the fundamentals of numerical approximation for ODEs and PDEs are the book due to Leveque (also used in DTU Course 02687). Remark, it is also possible to learn hidden physics within stochastic differential equations (SDEs), e.g. see O'Leary et al. (2021).

In the following, we focus on mastering basic techniques through working out some examples.

Boundary Value Problem

Consider a boundary value problem (BVP) stated in the general form

$$\begin{aligned} \mathcal{L}(u) &= f \quad \text{in} \quad \Omega \\ \mathcal{B}(u) &= g \quad \text{on} \quad \partial\Omega \end{aligned}$$

on the domain $\Omega \cong \mathbb{R}^d$, where d is the spatial dimension and $\mathcal{L}$ is a linear second-order, elliptic differential operator and $\mathcal{B}$ is an operator that defines the boundary conditions.

To define a PINN that solves the BVP the input data collocation points located inside the domain Ω and on the boundary $\partial\Omega$ are used to define where the BVP is to be satisfied. A feedforward neural network $\mathcal{N}(x;\Theta)$ is trained to satisfy the BVP in a least square sense such that all parameters Θ (weights and biases) of the neural network is determimed. The loss function is constructed such that point-wise residual errors of the PDE is minimized at the collocation points. The loss function can be defined as

$$\mathcal{M}(\Theta) := \mathcal{M}_{\Omega}(\Theta) + \mathcal{M}_{\partial\Omega}(\Theta)$$

where

$$\begin{aligned} \mathcal{M}_{\Omega}(\Theta) &:= \frac{1}{N_f}\sum_{i=1}^{N_f} |\mathcal{L}(\mathcal{N}(x_i^f;\Theta)) - f(x_i^f)|^2 \\ \mathcal{M}_{\partial\Omega}(\Theta) &:= \frac{1}{N_g}\sum_{i=1}^{N_g} |\mathcal{B}(\mathcal{N}(x_i^g;\Theta)) - g(x_i^g)|^2 \end{aligned}$$

Collocation points x_i^f, $i = 1, ..., N_f$ are located in the domain Ω and $x_i^g, \quad i = 1, ..., N_g$ are located on the boundary $\partial\Omega$. The derivatives of the neural network in the operators are evaluated using the backward propagation (backprop) algorithm[8] and computed using automatic differentiation (AD) that is available in the machine learning frameworks. This is exploited in PINN approaches to evaluate the residuals that is used to define the loss terms during each epoch of the training phase. The training procedure for the neural network corresponds to solving the optimization problem

$$\Theta^* := \arg\min_{\Theta} \mathcal{M}(\Theta)$$

[8]Rumelhart, Hinton & Williams (1986). "Learning representations by back-propagating errors" (PDF). Nature. 323 (6088): 533–536.

using a stochastic gradient approach based on mini-batches built from all the collocation points. The outcome of the optimization procedure is the set of coefficients Θ that defines the parameters of the neural network. After successful training, the neural network will be able to represent the solution across all parts of the parameter space covered in the training, and hence it will be possible to 'interpolate' between the collocation points used in the training procedure.

f) Setup a PINN approach to solve a Poisson problem $\nabla^2 u = f(x)$ with exact solution $u(x)$ that you choose and from which you can calculate $f(x)$ and the boundary conditions. Try and determine a Neural Network $\mathcal{N}(x;\Theta) \approx u(x)$. That is, sample the coordinates x in the domain Ω and on it's boundaries $\partial\Omega$ and feed these to the network during training. Try and do this both in one ($d = 1$) and two ($d = 2$) spatial dimensions.

Remark, a common problem in training of PINNs with multiple loss terms is that the loss terms may have different magnitude. A way to deal with this is to introduce adaptive weight loss balancing of the loss terms during the training procedure. In the above example, this suggest introducing a scalar weight $\lambda > 0$ that help balance the losses to be similar size for the optimization to perform effectively

$$\mathcal{M}(\Theta) := \lambda \mathcal{M}_{\Omega}(\Theta) + \mathcal{M}_{\partial\Omega}(\Theta)$$

g) Modify your PINN code to include adaptive weight loss balancing, e.g. by implementing SoftAdapt (See also the Github repos with code).

Initial Value Problems of ODE type

Consider an initial value problem (IVP) stated in the general form

$$\begin{aligned} u'(t) &= \mathcal{F}(t,u) \quad \text{in} \quad \mathcal{T} \\ u(t_0) &= u_0 \end{aligned}$$

on the domain $\Omega \cong \mathbb{R}$, $\mathcal{T} = [t_0, t_{final}]$ where $\mathcal{F}$ is a Lipschitz continuous function and the last equation valid at $t = 0$ is the initial condition.

To define a PINN that solves the IVP the input data is the collocation points located inside the domain $\mathcal{T}$. A feedforward neural network $\mathcal{N}(t;\Theta)$ is trained to satisfy the IVP in a least square sense such that all parameters Θ (weights and biases) of the neural network is determined. The loss function is constructed such that point-wise residual errors $\mathcal{R} = \mathcal{F}(t,u) - u'(t)$ of the ODE is minimized. The loss function can be defined as

$$\mathcal{M}(\Theta) := \mathcal{M}_{\Omega,ODE}(\Theta) + \mathcal{M}_{\Omega,IC}(\Theta)$$

where

$$\begin{aligned} \mathcal{M}_{\Omega,ODE}(\Theta) &:= \frac{1}{N_f} \sum_{i=1}^{N_f} |\mathcal{F}(\mathcal{N}(t_i^f;\Theta)) - \mathcal{N}'(t_i^f)|^2 \\ \mathcal{M}_{\Omega,IC}(\Theta) &:= \frac{1}{N_i} \sum_{i=1}^{N_i} |\mathcal{N}(t_0^i;\Theta) - u(t_0)|^2 \end{aligned}$$

Collocation points (t_i^f), $i = 1, ..., N_f$ are located in the domain $\mathcal{T}$ and (t_i^i), $i = 1, ..., N_i$ are located across the domain $\mathcal{T}$ at initial time t_0. The derivative of the neural network in the loss function can be evaluated using the backward propagation algorithm and computed using automatic differentiation that is available in the machine learning frameworks.

h) Setup a PINN to solve the test equation $\mathcal{F}(u; \lambda) = \lambda u$, $\lambda \in \mathbb{R}$ with exact solution $u(t; \lambda) = u_0 \exp(\lambda t)$. The solution either decays, grows or stays constant depending on the value of λ. Try and determine a Neural Network $\mathcal{N}(t, \lambda; \Theta) \approx u(t, \lambda)$ such that the parameter λ in the problem is parametrized on an interval for quick estimation of the solution curves for different λ after training. That is, sample the coordinates (t, λ) in the parameter domain $\mathcal{T} \times \Omega_\lambda$ and feed these to the network during training. Compare the predicted values and the exact solutions on a time interval $\mathcal{T} = [0, 2]$ for $\lambda \in [-1, 1]$.

i) Setup a PINN to solve the problem for a damped harmonic oscillator

$$x'' = \omega_0^2 - \lambda x', \quad t > t_0$$

The initial conditions can be defined as $x(t_0) = 1$, $v(t_0) = 0$ and the parameters selected as $\omega_0 = 3$ and $\lambda = 0.5$. Try and solve first in a time interval of $0 < t < 10$. Then try and solve the problem numerically to obtain data points for the solution. Then, try and utilize from a single up to multiple data points from within the range $0 < t < 5$ to train the PINN model, by incorporating an additional loss term for the data points that supplements the loss term for the differential equations(s) and the initial condition(s). Does adding data points help the PINN train faster? This example should demonstrate that you can find the solutions using PINN with no data points and also benefit from utilizing data points. If time permits, try and see how fast does the PINN approach converge to the solutions vs. using a purely data-driven approach based on the numerical data points for the solution in the same time range? Try and use to the PINN framework to extrapolate outside the range of the data-driven approach and compare the neural networks found.

Initial Value Problems of PDE type

Consider an initial value problem (IVP) stated in the general form

$$\begin{aligned} u_t = \mathcal{L}(u) \quad & \text{in} \quad \Omega \times \mathcal{T} \\ \mathcal{B}(u) = g(t) \quad & \text{on} \quad \partial\Omega \times \mathcal{T} \\ u(x, t_0) = h(x) \quad & \text{in} \quad \Omega \end{aligned}$$

on the domain $\Omega \cong \mathbb{R}^d$, $\mathcal{T} = [t_0, t_{final}]$ where d is the spatial dimension and $\mathcal{L}$ is a linear differential operator and $\mathcal{B}$ is an operator that defines the boundary conditions and the last equation valid at $t = 0$ is the initial condition.

To define a PINN that solves the IVP the input data is the collocation points located inside the parameter domain Ω and on the boundary $\partial\Omega$. A feedforward neural network $\mathcal{N}(x, t; \Theta)$ is trained to satisfy the IVP in a least square sense such that all parameters Θ (weights and biases) of the neural network is determimed. The loss function is constructed such that point-wise residual errors $\mathcal{R} = \mathcal{L}(u) - u_t$ of the PDE is minimized. The loss function can be defined as

$$\mathcal{M}(\Theta) := \mathcal{M}_{\Omega,PDE}(\Theta) + \mathcal{M}_{\partial\Omega}(\Theta) + \mathcal{M}_{\Omega,IC}(\Theta)$$

where

$$\mathcal{M}_{\Omega,PDE}(\Theta) := \frac{1}{N_f}\sum_{i=1}^{N_f} |\mathcal{L}(\mathcal{N}(x_i^f, t_i^f; \Theta)) - \partial_t \mathcal{N}(x_i^f, t_i^f)|^2$$

$$\mathcal{M}_{\partial\Omega}(\Theta) := \frac{1}{N_g}\sum_{i=1}^{N_g} |\mathcal{B}(\mathcal{N}(x_i^g; \Theta)) - g(x_i^g, t_i^g)|^2$$

$$\mathcal{M}_{\Omega,IC}(\Theta) := \frac{1}{N_i}\sum_{i=1}^{N_i} |\mathcal{N}(x_i^i, t_0^i; \Theta) - u(x_i^i, t_0)|^2$$

Collocation points (x_i^f, t_i^f), $i = 1, ..., N_f$ are located in the parameter domain $\Omega \times \mathcal{T}$ and (x_i^g, t_i^g), $i = 1, ..., N_g$ are located on the boundary $\partial\Omega$ and (x_i^i, t_i^i), $i = 1, ..., N_i$ are located across the domain Ω at initial time t_0. The derivatives of the neural network in the operators are evaluated using the backward propagation algorithm and computed using automatic differentiation that is available in the machine learning frameworks.

i) Setup a PINN to solve a linear advection equation $\mathcal{L} = -cu_x$, $c > 0$ with exact solution $u(x,t) = h(x - ct)$ that you choose by defining $h(x)$ and from which you can calculate the boundary conditions and the initial values. This system describes wave propagation from left to right when $c > 0$ and hence a boundary condition is needed at the upwind (=opposite direction of wave propagation direction) boundaries in a finite domain, and alternatively, a periodic domain can be assumed by imposing $u(x_a, t) = u(x_b, t)$ assuming $x \in [x_a, x_b]$. Try and determine a Neural Network $\mathcal{N}(x, t; \Theta) \approx u(x, t)$. That is, sample the coordinates (x, t) in the domain $\Omega \times \mathcal{T}$ and on it's left boundary $\partial\Omega$ and feed these to the network during training. Try and do this in one $(d = 1)$ spatial dimension.

j) Setup a PINN to solve instead a nonlinear PDE equation stated as

$$u_t = \mathcal{N}(u), \quad \text{in} \quad \Omega \times \mathcal{T}$$

where $\mathcal{N}(u)$ is a nonlinear differential operator, e.g. for the Korteweg–De Vries (KdV) equation describing traveling wave evolution, the operator can be defined as $\mathcal{N}(u) = -6u\partial_x u - \partial_{xxx} u$. What changes are needed to the PINN approach to handle this change? Solve the KdV equation using the exact soliton solution $u(x, t) = h(x - ct)$ with
$h(x) = \frac{1}{2}\text{sech}^2(\frac{1}{2}\sqrt{c}(x - x_0))$, $c > 0$, and where x_0 is the center of the soliton at time $t = 0$.