Physics Informed Neural Networks for Solving PDEs (Direct and Inverse Problems)

Replicating the results of the paper: "DeepXDE: A Deep Learning Library for Solving Differential Equations" by Lu Lu, Xuhui Meng, Zhiping Mao and George Em Karniadakis.

POLITECNICO
MILANO 1863

Motivation

Differential equations, both ordinary and partial, are one of the most important modeling tool in science, physics and engineering. Complex differential equations appear everywhere and the scientific community is always trying to find the best methods to solve those problems. Nowadays one of the most used technique is Finite Element Method (FEM) which discretize the domain of the independent variables an tries to build a solution point after point (mesh), starting from the initial conditions. This has been studied for decades and a lot of very efficient solvers have been developed for commercial usage, so this method is now the standard choice. Unfortunately, this approach presents some problems, the main one being the mesh which might be extremely complex, especially for high dimensional problems (curse of dimensionality). Moreover, we have only talked about direct problems (simulation), but often times we don't know all the parameters of the system and we want to learn those from the data, which are inherently noisy. Doing this with classical FEM methods is very inefficient since we might find ourself running a lot of different simulations with different parameters or boundary conditions. The last problem of FEM is that we will always find a discretized solutions while, instead, we are solving a continuous problem.

All those issues, together with the increasing need of precise simulations for differential problem led to the birth of Physics-informed neural networks (PINN), a very promising field of Scientific Machine Learning (SciML).

Architecture

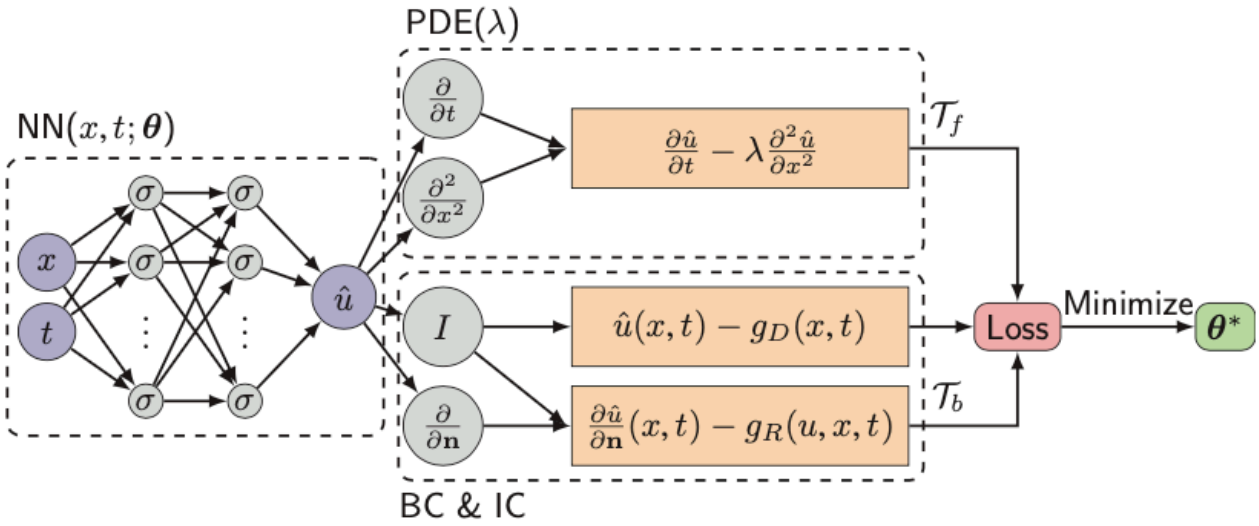
PINN are very simple from an architectural point of view, since the main innovation stands in the way we compute the loss function. Usually we use an ANN with FeedForward architecture, tanh activation function (or even sinusoidal if the solution is strongly periodical) and a combination of ADAM and BFGS optimizer, taking advantage of Automatic Differentiation for computing the gradient of the loss with respect to the parameters of the network. The formulas to compute the value of each node in each layer is the following:

Let $\mathcal{N}^L(\mathbf{x}) : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$ be an L-layer neural network, or an $(L-1)$-hidden layer neural network, with N_ℓ neurons in the ℓth layer ($N_0 = d_{\text{in}}$, $N_L = d_{\text{out}}$). Let us denote the weight matrix and bias vector in the ℓth layer by $\boldsymbol{W}^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and $\mathbf{b}^\ell \in \mathbb{R}^{N_\ell}$, respectively. Given a nonlinear activation function σ, which is applied elementwisely, the FNN is recursively defined as follows:

$$
\begin{aligned}
\text{input layer:} \quad & \mathcal{N}^0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{d_{\text{in}}}, \\
\text{hidden layers:} \quad & \mathcal{N}^\ell(\mathbf{x}) = \sigma(\boldsymbol{W}^\ell \mathcal{N}^{\ell-1}(\mathbf{x}) + \boldsymbol{b}^\ell) \in \mathbb{R}^{N_\ell} \quad \text{for} \quad 1 \le \ell \le L-1, \\
\text{output layer:} \quad & \mathcal{N}^L(\mathbf{x}) = \boldsymbol{W}^L \mathcal{N}^{L-1}(\mathbf{x}) + \boldsymbol{b}^L \in \mathbb{R}^{d_{\text{out}}};
\end{aligned}
$$

Loss Function

As we said before the main innovation of PINN is the loss function, which is basically interpolating the datapoints while being regularized by the residual of the differential equation. The loss is composed of 2 main components: the first one is the L2 norm of the PDE residuals and the second one is the L2 norm of the error on the initial conditions (IC) or boundary conditions (BC), summed together. It is important to define proper weights to this summation, since we might risk to learn only a specific solution (like a steady state) of a random solution without respecting IC and BC. Moreover, we should also note that weighting the 2 losses is relevant because often times those 2 components don't even have the same unit of measure. The general architecture is the following:



For computing the loss we need 2 dataset of points. The first one contains a sample of the inner domain (where we compite the PDE residual) and the second one contains a sample of the boundary (where we compute the MSE on the known values). The loss function, as said before, can be expressed in the following way:

$$\mathcal{L}(\boldsymbol{\theta};\mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta};\mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta};\mathcal{T}_b), \tag{2.2}$$

where

$$\mathcal{L}_f(\boldsymbol{\theta};\mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x}\in\mathcal{T}_f} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right\|_2^2,$$

$$\mathcal{L}_b(\boldsymbol{\theta};\mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x}\in\mathcal{T}_b} \|\mathcal{B}(\hat{u},\mathbf{x})\|_2^2,$$

In this representation we can see the IC are treated as a particular case of boundary conditions, so that they can be unified. This approach is very general and it can describe IDE, ODE and PDE with Neumann, Dirichlet and Robin conditions. We will see an example of all those cases. Sadly, this function has the problem of competing loss terms, which is difficult to solve since the weights are typically hyper parameters and we have to tune them.

Looking at the loss function the first issue we think about is the fact the we must differentiate the neural network with respect to the input, not the parameters. This turns out to not be a huge problem, since we can stil exploit automatic differentiation, expanding the computational graph to extract all the derivatives we need. Those derivatives will be exact since we don't need to resort to numerical differentiation. Anyway, this doesn't mean that computing all those gradients will be free, in fact, the majority of the time spent during training will be devoted to exactly that. That's because there's a huge amount of partial derivatives to be computed, especially for higher order terms.

Advantages of PINNs and Inverse Problem

The advantages of PINNs are immediately evident. First of all we don't need a complex geometrical mesh, we just a need a grid of points pseudo-randomly taken from the domain, we will also discuss a strategy to intelligently choose those points. Moreover, the neural network we'll find will be a functional approximation of the solution, which is exactly what we expect from a continuous problem. The last and most important advantage is that PINN can solve inverse problem as easily as direct ones.

We can just set the missing parameters as additional learnable parameters and the network will learn them together with the PDE solution. Obviously, we also need another term in the loss, which is composed by a dataset of known points besides IC and BC. The network can then use those points to figure out the best solution, discarding the noise and capturing the differential relationship together with the PDE parameters. We should also note that it's not easy to define how many points are necessary for this learning process, since a small amount of very noisy points could lead, potentially, to an ill posed problem. This is clear looking at the modified loss (everything else remains the same):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}) = w_f \mathcal{L}_f(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_f) + w_b \mathcal{L}_b(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_b) + w_i \mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i),$$

where

$$\mathcal{L}_i(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T}_i) = \frac{1}{|\mathcal{T}_i|} \sum_{\mathbf{x} \in \mathcal{T}_i} \|\mathcal{I}(\hat{u}, \mathbf{x})\|_2^2.$$

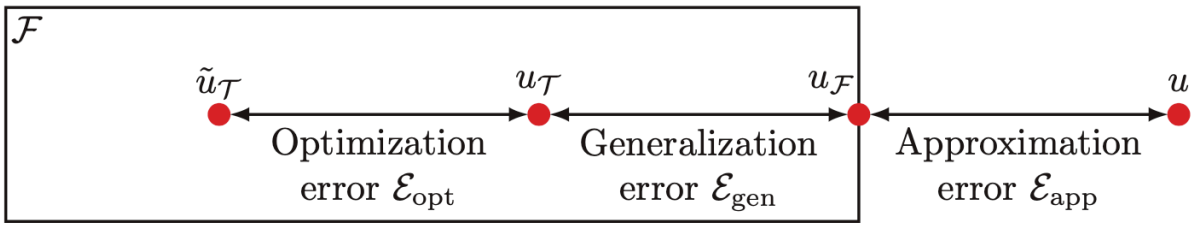
We then optimize $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$ together, and our solution is $\boldsymbol{\theta}^*, \boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}; \mathcal{T})$.

Approximation Theory

PINNs are not a magic solution, for instance there is the issue already mentioned of competing loss terms which is a problem in the minimizing process. Anyway, the real problem is error estimation which is de-facto non existing at this moment. This problem is related to the black-box nature of neural networks, and as we all know explainability is one the main area of open research in deep learning. On the other hand, FEM methods are clearly explainable and this, together with the optimization efficiency, is why PINN will probably not replace FEM methods for simulations and direct problems. That can't be said for inverse problems where PINN are, instead, very promising. Here I'll provide an abstract analysis of the sources of uncertainty:

We can then decompose the total error $\mathcal{E}$ as [11]

$$\mathcal{E} := \|\tilde{u}_{\mathcal{T}} - u\| \leq \underbrace{\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|}_{\mathcal{E}_{\text{opt}}} + \underbrace{\|u_{\mathcal{T}} - u_{\mathcal{F}}\|}_{\mathcal{E}_{\text{gen}}} + \underbrace{\|u_{\mathcal{F}} - u\|}_{\mathcal{E}_{\text{app}}}.$$



The Universal approximation theorem assures that a FeedForward Neural Network with enough neurons can simultaneously approximate any function and its partial derivatives. This is stated in an important theorem explained in the reference paper.

However, neural networks in practice have limited size. Let $\mathcal{F}$ denote the family of all the functions that can be represented by our chosen neural network architecture. The solution u is unlikely to belong to the family $\mathcal{F}$, and we define $u_{\mathcal{F}} = \arg\min_{f \in \mathcal{F}} \|f - u\|$ as the best function in $\mathcal{F}$ close to u (Figure 3). Because we only train the neural network on the training set $\mathcal{T}$, we define $u_{\mathcal{T}} = \arg\min_{f \in \mathcal{F}} \mathcal{L}(f; \mathcal{T})$ as the neural network whose loss is at global minimum. For simplicity, we assume that u, $u_{\mathcal{F}}$, and $u_{\mathcal{T}}$ are well defined and unique. Finding $u_{\mathcal{T}}$ by minimizing the loss is often computationally intractable [10], and our optimizer returns an approximate solution $\tilde{u}_{\mathcal{T}}$.
The approximation error $\mathcal{E}_{\text{app}}$ measures how closely $u_{\mathcal{F}}$ can approximate u. The generalization error $\mathcal{E}_{\text{gen}}$ is determined by the number/locations of residual points in $\mathcal{T}$ and the capacity of the family $\mathcal{F}$. Neural networks of larger size have smaller approximation errors but could lead to higher generalization errors, which is called bias-variance tradeoff. Overfitting occurs when the generalization error dominates. In addition, the optimization error $\mathcal{E}_{\text{opt}}$ stems from the loss function complexity and the optimization setup, such as learning rate and number of iterations. However, currently there is no error estimation for PINNs yet, and even quantifying the three errors for supervised learning is still an open research problem [36, 35, 25].

PINNs versus FEM

This is a confrontation table to better understand the fundamental difference between the 2 methods. As we can see they are completely different in both how they approach the problem and what result they provide.

	PINN	FEM
Basis function	Neural network (nonlinear)	Piecewise polynomial (linear)
Parameters	Weights and biases	Point values
Training points	Scattered points (mesh-free)	Mesh points
PDE embedding	Loss function	Algebraic system
Parameter solver	Gradient-based optimizer	Linear solver
Errors	$\mathcal{E}_{\text{app}}$, $\mathcal{E}_{\text{gen}}$, and $\mathcal{E}_{\text{opt}}$ (subsection 2.4)	Approximation/quadrature errors
Error bounds	Not available yet	Partially available [14, 26]

Residual Adaptive Sampling (RAR)

Now we'll focus on how to choose the grid of points on which we will compute the residuals, the most critical loss term for a Physics-informed neural network. Randomly choosing those points is probably not the best idea since the real function is not equally smooth on all the domain and we prefer to have a lot of points where the solution function has the highest gradient. If we know, more or less, how the gradient is distributed along the domain we can pick manually the most effective points, where the function is changing more rapidly, hoping that it will be enough for the PINN to capture the solution. Unfortunately, this is rarely the case. Instead, we can use the RAR method (Residual Adaptive Refinement) to automatically sample the points where my network is currently making the biggest mistake. First of all we check if we need to add any new point, computing the integral of the residuals on all the domain (we use Monte-Carlo integration, analytically intractable). Then, if that value is over a certain threshold we add the N points with the worst residual error and we repeat until convergence. The calculations are the following:

$$\mathcal{E}_r = \frac{1}{V}\int_\Omega \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right\| d\mathbf{x}$$

where V is the volume of Ω.

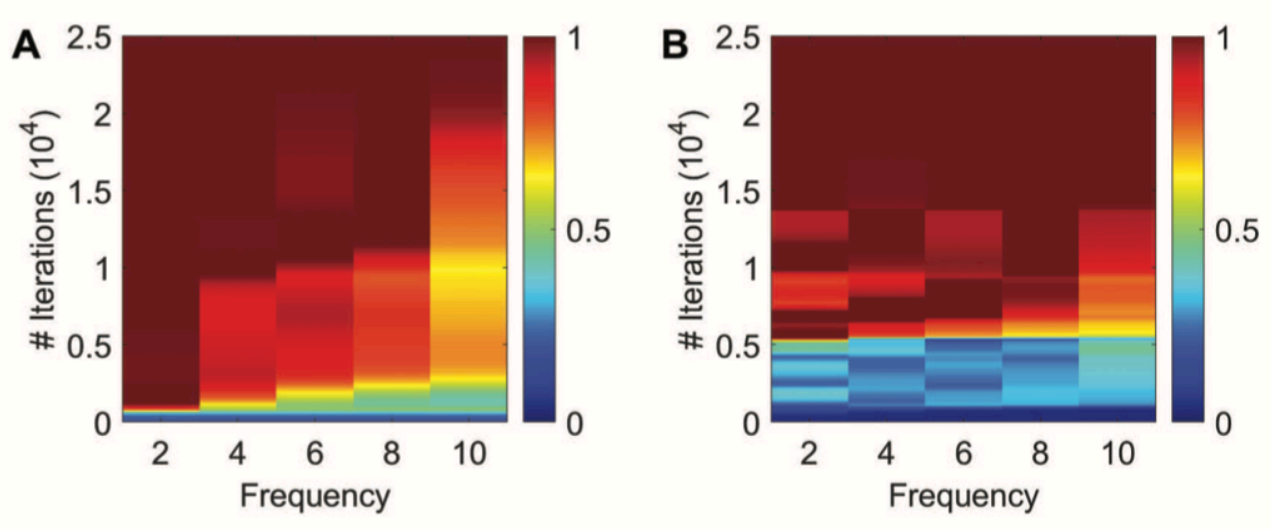
Estimate the mean PDE residual $\mathcal{E}_r$ in (2.3) by Monte Carlo integration, i.e., by the average of values at a set of randomly sampled dense locations $\mathcal{S} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\mathcal{S}|}\}$:

$$\mathcal{E}_r \approx \frac{1}{|\mathcal{S}|}\sum_{\mathbf{x}\in\mathcal{S}} \left\| f\left(\mathbf{x}; \frac{\partial \hat{u}}{\partial x_1}, \ldots, \frac{\partial \hat{u}}{\partial x_d}; \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 \hat{u}}{\partial x_1 \partial x_d}; \ldots; \boldsymbol{\lambda}\right)\right\|.$$

Hard constrains are an useful technique to be used when the IC and BC are simple and they can be enforced by a small modification in the last layer of the network. We have to formulate a guess (ANSATZ) on a reasonable form of the solution. We then apply our surrogate model on the last layer of the PINN, making sure that all the points contained in IC and BC are forced to take a specific value. This is great because it gets rid of the competing loss terms and it lets the network focus only of the differential residuals. The issue is that often times it's not easy (or not possible) to find a nice guess function where we can plug in the last layer of the PINN. In that case we must resort to classical soft constrains. Here we see an example of this approach:

$$\hat{u}(x) = x(x-1)\mathcal{N}(x) \quad \text{enforces} \quad u(0) = u(1) = 0 \text{ with } \Omega = [0,1]$$

Fourier analysis of the neural networks is very interesting to do during training, since it gives us an understating of which frequencies are learned by our model, depending on the training stage. For instance, we know that standard neural networks for functional approximation learn the the low frequencies before the high one. That might be an advantage because it acts as a sort of regularization, but it's also a disadvantage since it might slow down the training process. The learning mode of PINNs is different due to the existence of high-order derivatives and they seem to learn all the frequencies almost simultaneously. This is an heat map visualizing the main frequencies:



Integro-differential equations (IDEs) can also be solved. We still employ the automatic differentiation technique to analytically compute all the derivatives, while we approximate the integral operators numerically using classical methods, such as Gaussian quadrature. Therefore, we introduce a fourth error component, the discretization error. This is an example of integral discretization applied to the Volterra's equation.

$$\frac{dy}{dx} + y(x) = \int_0^x e^{t-x} y(t) dt,$$

we first use Gaussian quadrature of degree n to approximate the integral

$$\int_0^x e^{t-x} y(t) dt \approx \sum_{i=1}^{n} w_i e^{t_i(x)-x} y(t_i(x)),$$

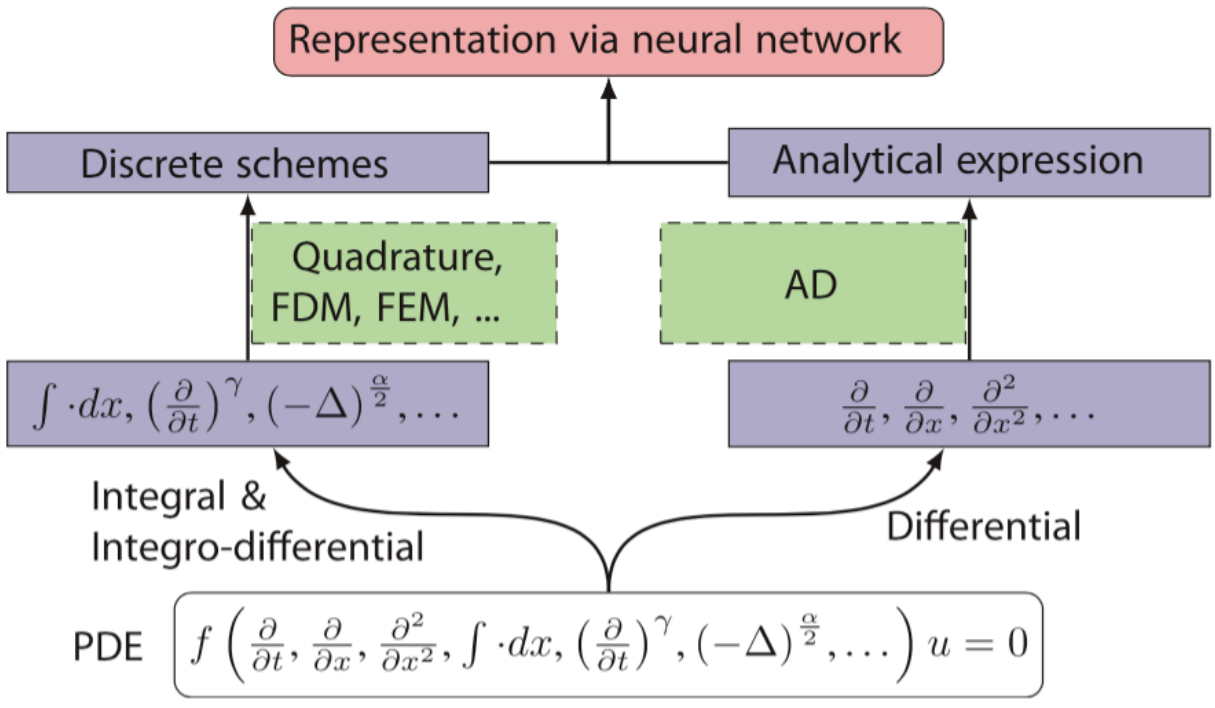
and then we use a PINN to solve the following PDE instead of the original equation:

$$\frac{dy}{dx} + y(x) \approx \sum_{i=1}^{n} w_i e^{t_i(x)-x} y(t_i(x)).$$

The final equation is a standard ODE where we have a numerical constant instead of the integral. The DeepXDE library uses the Gaussian quadrature, instead, I will use the trapezoid rule in my practical implementations. This can be applied to any type of IDE which will be then treated as a normal ODE.

Modelling

Here we can see a final and general modeling schema for PINNs:



Those machine learning problems can be modeled as unsupervised (hard constrains), semi-supervised (direct problem) and supervised (inverse problem). This contributes in presenting the fact the PINN are a very general and versatile model with many applications in Scientific Machine Learning.

Implementation

I decided to implement PINNs in 2 different ways for 3 different problems. The first way is using the DeepXDE library, which was described in the paper and the second one is using Jax, Numpy and Scipy so that I could really understand and personalize the PINN, building it from scratch. I coded the network architecture, the optimization algorithm and the loss function trying to come up with my personal implementation as much as possible. Numpy was necessary for linear algebra, Scipy for optimizing line search and for integrating with trapezoid method. Finally I used JAX to compute gradients efficiently with automatic differentiation.

As expected the DeepXDE version is a little more precise and a lot faster (especially for higher order domain), but my personal solutions are still pretty good and allowed me to clearly understand PINNs. For more complex problem, especially for more complex domain, the DeepXDE library is extremely useful thanks to the possibility of building the domain with Constructive Solid Geometry, combing simple shapes into complex ones.

Overall, I liked DeepXDE and I find it versatile and clear since it's very similar to the mathematical formulation, but I also enjoyed building my own PINNs.

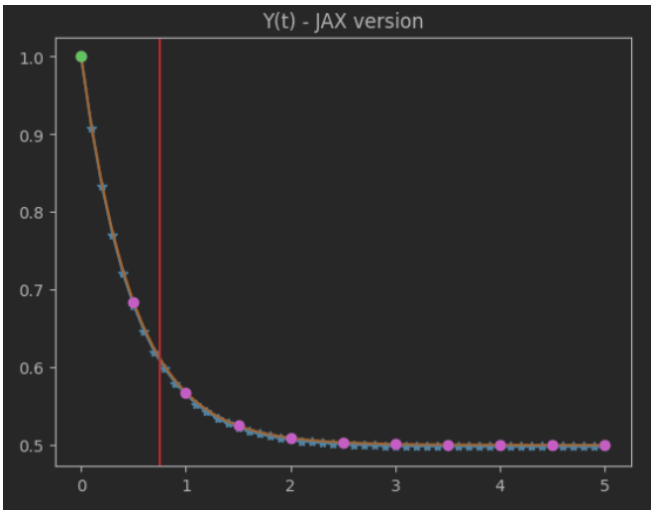
The results, as we will see, are satisfying for both the approaches.

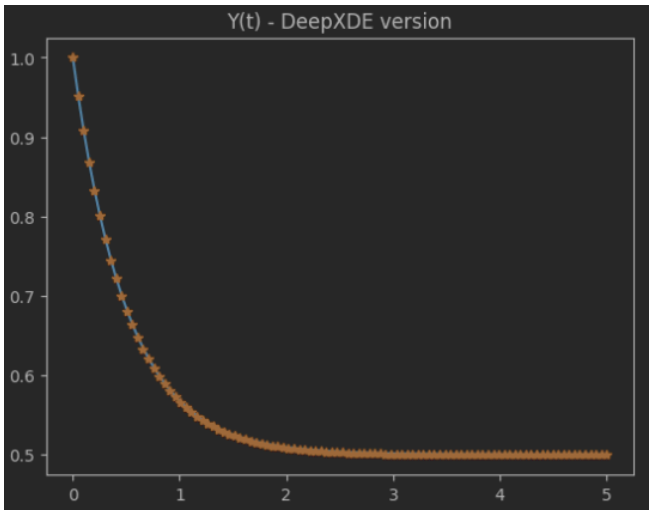
I provided 6 notebooks for this project, 3 for my personal implementation in JAX and 3 with the DeepXDE library. I am also providing, as a pickle file, the trained parameters and the final grid (after RAR method) for the Burger equation, since the training took around 1 hour. I have even saved the learning process of the variable C (inverse problem) in the Kirchhoff equation, thanks to a nice feature of the DeepXDE library. Even tough my implementations had good results I would definitely use DeepXDE since it's much faster, both to code and to train. Moreover, the Constructive Solid Geometry (CSG) is very convenient for real problems where the domain is often times elaborate and irregular. The only disadvantage is that we lose a bit of control and knowledge of what is happening under the hoods, but that's completely normal when using any kind of library.

Volterra Equation for Ruin Theory

$$\frac{dy}{dx} + y(x) = \int_0^x e^{t-x} y(t) dt, \quad y(0) = 1,$$

- Network Architecture: FFNN network, tanh activation, layers [1, 16, 8, 4, 1]
- Residual points: grid sampling on 1D domain [0, 5] optimized with BFGS
- Notable Choice: hard constrains and trapezoid rule for integration





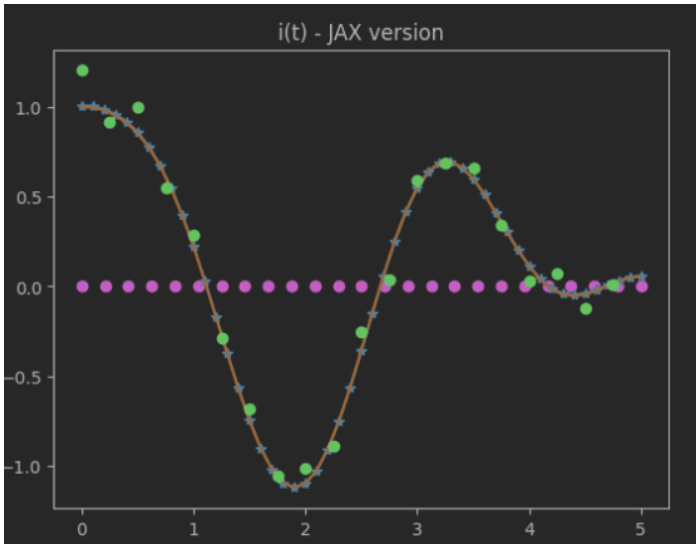
This problem is the simplest one, but it's still interesting since it involves both derivative and integrals. The training was very fast and effective even tough I used the trapezoid method instead of Gaussian quadrature which is obviously less precise. The relative error in x = 0.75 (red line) is less the 0.5% and that's remarkable since that seems to be the most critical region for the solution. This nice result is also obtained thanks to hard constrains, enforced by multiplying the last layer by the input 't' and then adding 1. This approach lets the network focus on the differential relationship with the first order derivative and we are able to learn the true solution almost perfectly. We could probably reach an even higher accuracy by employing a more precise approximation for integrals, such as the Simpson-Cavalieri method (parabolic curve), but this result was good enough for my case.

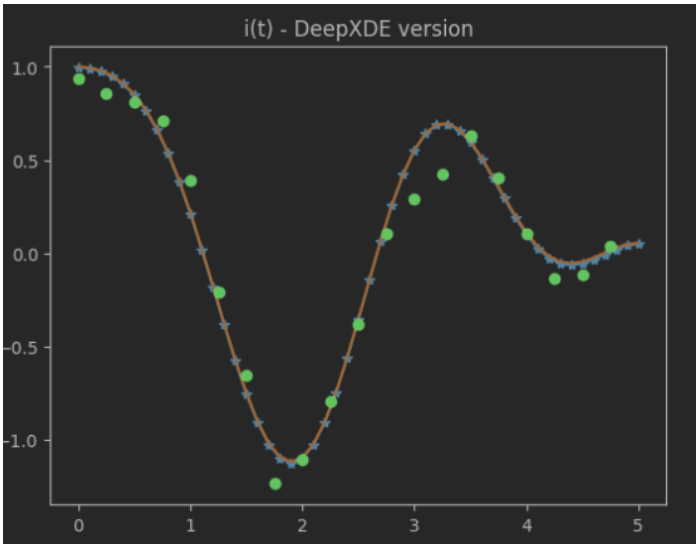
As we can see the obtained solution is, almost exactly, the one we expect.

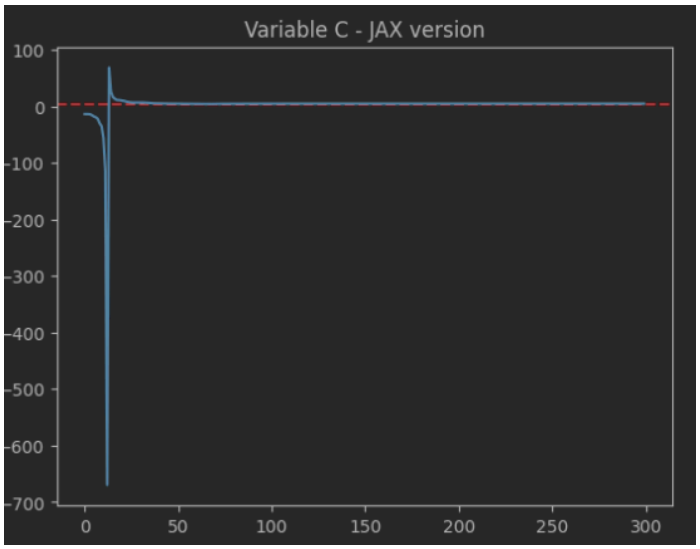
Kirchhoff Equation for RLC Circuit - Inverse Problem

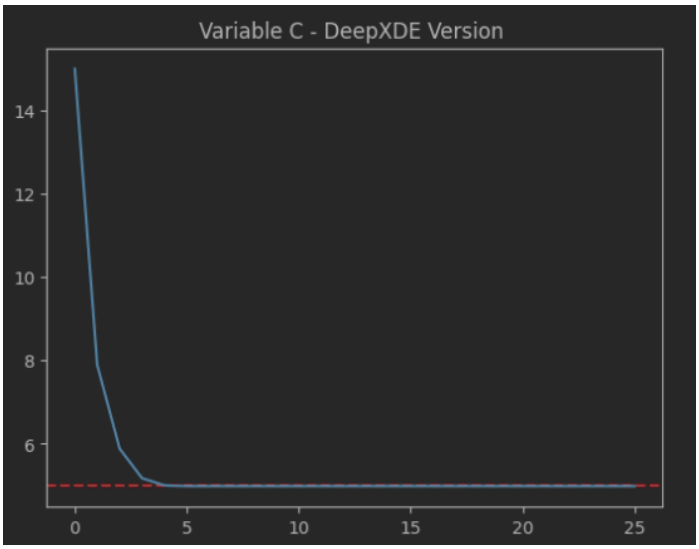
$$y'' + 0.5\,y' + 3\,y = 2\,\cos(3\,x)\,,\quad x_0 = 0,\ y(0) = 1,\ y'(0) = 0$$

- Network Architecture: ResNet network, tanh activation, layers [1,12,12,12,1]
- Residual points: grid sampling on 1D domain [0, 5] optimized with BFGS
- Notable Choice: ResNet architecture and Inverse Problem for the capacitor



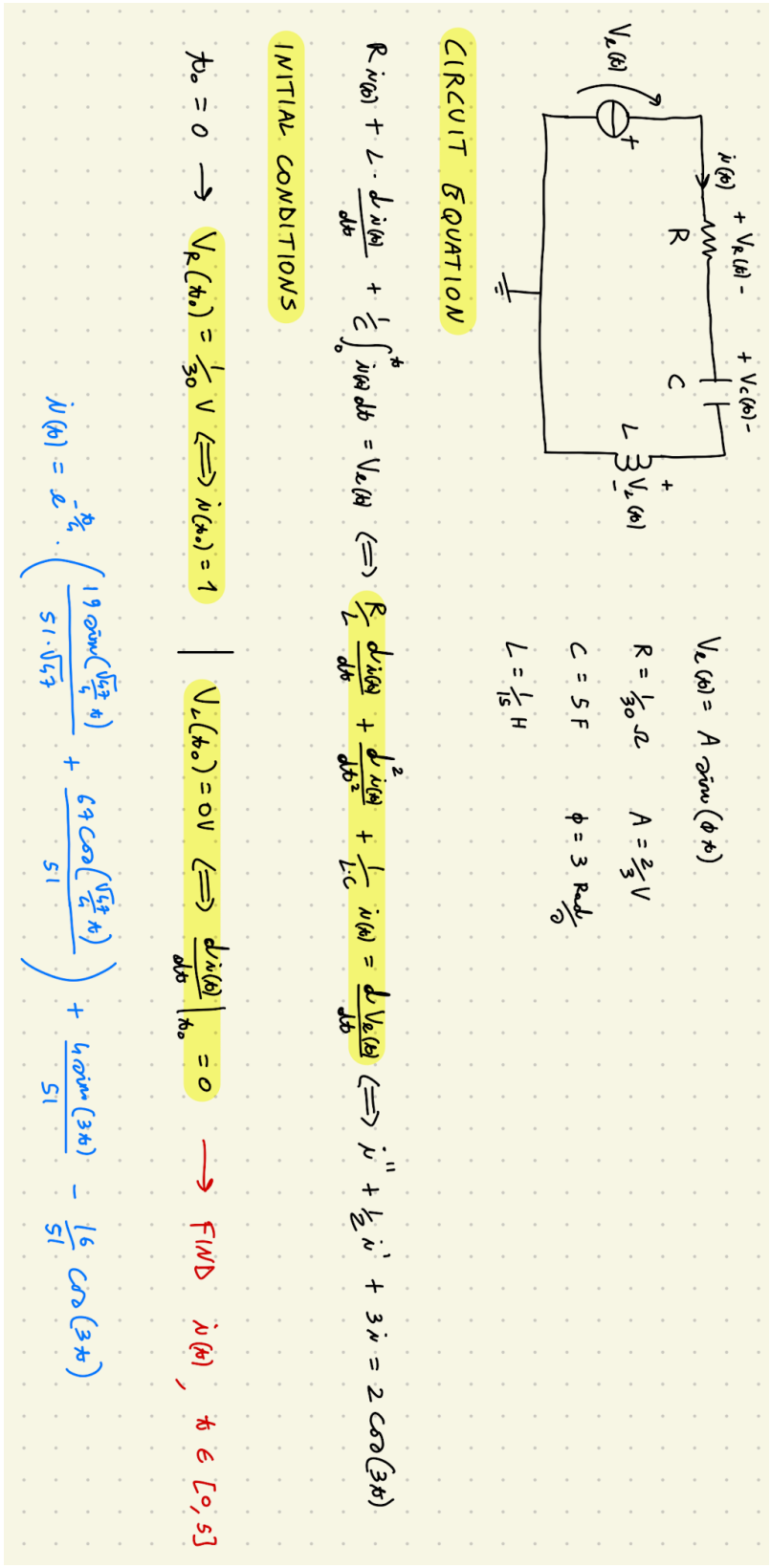






This equation represent the solution of a RLC circuit which is presented in the following page. The solution function is the current flowing in the circuit.

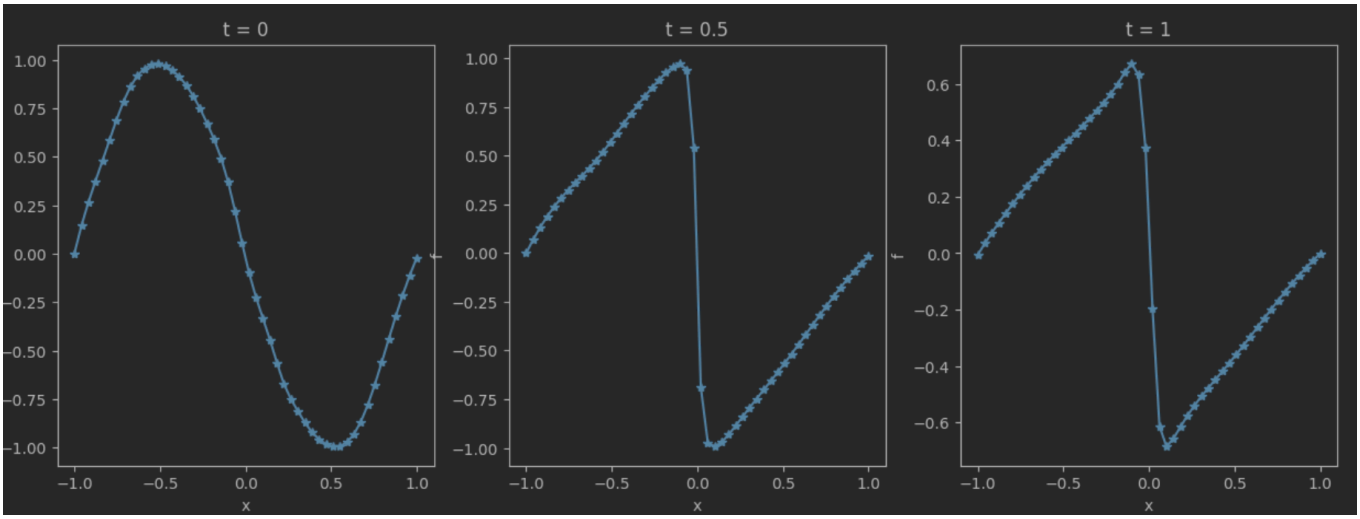
This problem is the one I preferred since this second order equation is an Harmonic Oscillator and it can be applied to RLC circuits which I studied during my bachelor degree. The parameters and the general model for the problem can be seen below. In this problem I decided to set the capacity C as an unknown parameter, so that the PINN will have to learn it together with the functional solution. The capacity is learned correctly but the process is strongly non-monotonic a typical problem of PINN networks. The points used for the inverse problem are sampled from the real analytical solution, with a random gaussian noise added to it. In this case, I decided to weight those latter points half of the rest since I don't want the network to learn the noise and possibly overfit the data. Finally, we can see that in both cases the function representing the electrical current is correctly learned and the final capacity is 5.05 Farad, very close to the real value of 5 Farad. This example is very important because it shows how to apply PINNs to inverse problems, a very common type of task which arises when parameters are very difficult to measure. As said before, it's the most promising application area for PINNs, currently.

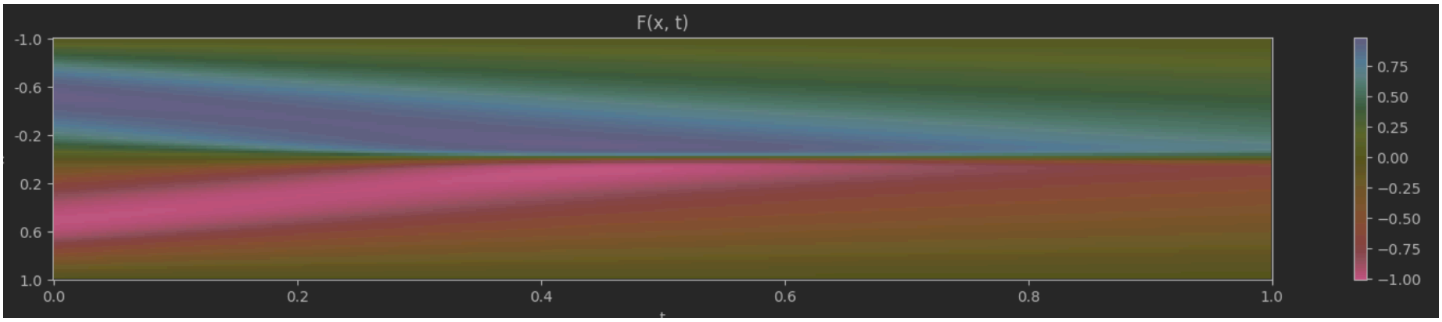


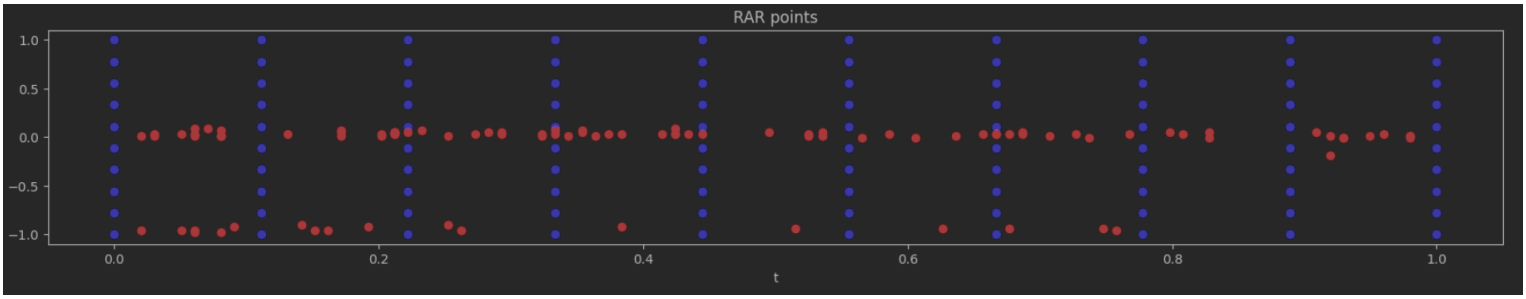
Burger Equation for Viscous Fluids

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \quad x \in [-1,1],\ t \in [0,1],$$
$$u(x,0) = -\sin(\pi x), \quad u(-1,t) = u(1,t) = 0.$$

- Network Architecture: FFNN network, tanh activation, layers [2, 16, 8, 4, 1]
- Residual points: RAR 2D domain [0,1]x[-1,1] optimized with ADAM + BFGS
- Notable Choice: residual-based adaptive refinement and soft constrains





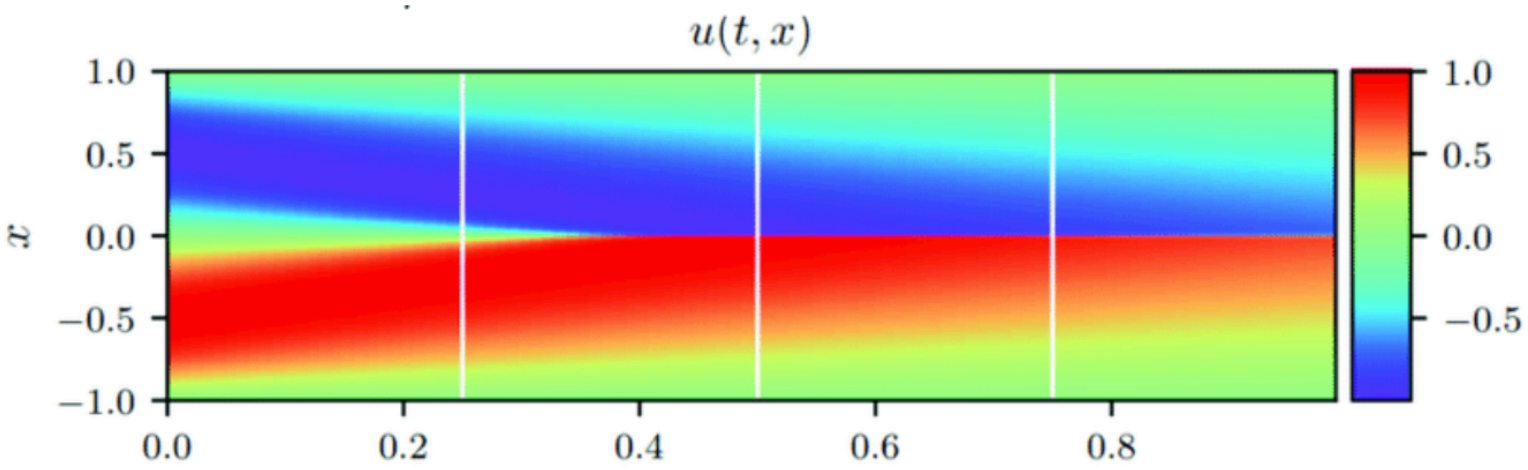


This was by far the most challenging problem, for a variety of reasons: the differential equation is partial, non-linear and the domain is 2-dimensional. For those reasons I decided to exploit the RAR method (Residual Adaptive Refinement) since the classical grid sampling or uniform sampling was not effective enough. I calculate the residual integral described before every 5 epochs and I keep adding the 10 most critical points found, starting from the 100 points obtained by a standard grid sampling. I repeat this process until convergence (integral value less than 0.01) or until I reach the maximum amount of points (set to 200, double the number of starting points).

For the optimization process, due to the complexity of the problem (with respect to the previous ones) I had to use a combination of ADAM and BFGS. Initially ADAM is useful to go directly towards a descending direction, without spending too much time in computation, then when it stops making real progresses BFGS starts and given its second order nature it's able to go deeper in the best minimum direction. Another positive factor is that with BFGS we don't need to specify a learning rate (very difficult in those cases), and that's one of reasons why BFGS is often preferred when training PINNs (the network is not too big or complex, usually).

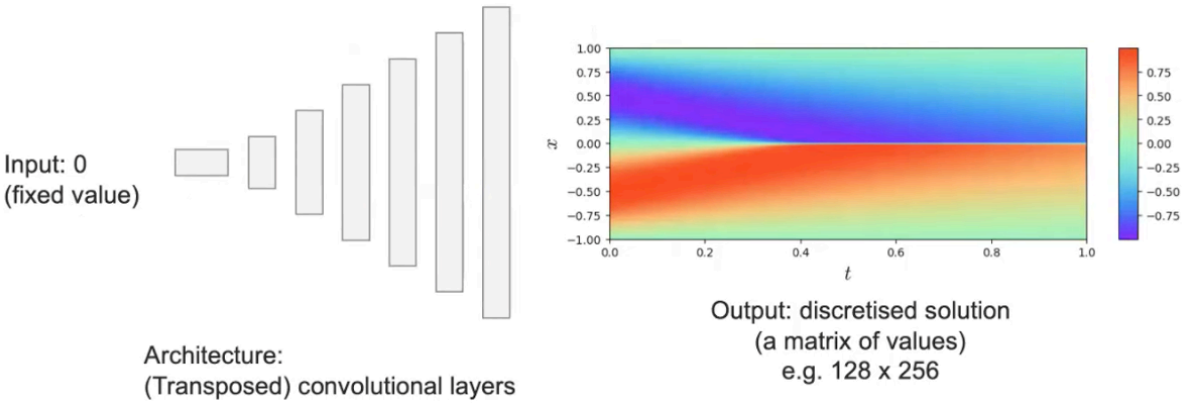
The first 3 plots represent the velocity, when the time is fixed and they show us that the PINN is able to correctly learn and model the impulse, which is very challenging even for FEM methods. That is thanks to the RAR sampling, which is shown in the last plot, and as expected the majority of the new points are placed in the middle where x (displacement) is zero. That region is exactly the one where the impulse takes place and the RAR method is able to correctly capture this behavior. It's important to note that training took 1 hour.

Finally, the second plot shows us the velocity of the fluid in the tube and it's very similar to the result obtained by FEM resolution, which is the following:



Extensions

A possibile extension of classical PINNs are Discretized PINNs. The idea is to use Convolutional Neural Networks to learn a discretized solution in the form of a matrix (image). The nice part is that the solution to a differential equation, visualized as a matrix, won't be random collection of pixels, instead, there will probably be a clear spatial correlation between all the pixels, like the one we see in the example on Burger Equation. This is convenient, since CNN are very good in finding and learning spatial correlation using kernels for matrix filtering. Note that we can't use AD to find the derivative of the network with respect to the inputs, because the input is now a fixed vector passing in various layers: upsampling, convolution, pooling and padding. We resort to Finite Difference method for differentiation, which is extremely easy since we already have a discretized solution (that's exactly what the network produces !). Obviously, we are still going to use Automatic Differentiation for updating the parameters.



$$NN(\theta)_{ij} \approx u(x = x_i, t = t_j) \qquad \text{Instead of: } NN(x, t; \theta) \approx u(x, t)$$

Those networks can also handle mixed discrete and continuous inputs. The new loss function becomes: (it can be computed by shifting the output matrix)

$$L_p(\theta) = \frac{1}{N_x N_t} \sum_{i}^{N_x} \sum_{j}^{N_t} \left(\frac{\delta NN(\theta)_{ij}}{\delta t} + NN(\theta)_{ij} \frac{\delta NN(\theta)_{ij}}{\delta x} - \nu \frac{\delta^2 NN(\theta)_{ij}}{\delta x^2} \right)^2$$

$$\frac{\delta NN(\theta)_{ij}}{\delta t} = \frac{NN(\theta)_{ij} - NN(\theta)_{i\,j-1}}{t_j - t_{j-1}}$$

$$NN(\theta)_{i\,j=0} = -\sin(\pi x_i)$$
$$NN(\theta)_{i=0\,j} = NN(\theta)_{i=128\,j} = 0$$

The BC are usually enforced as hard constrains, by fixing a part of the matrix. One immediate disadvantage is that using FD our derivatives are not precise. Moreover, if we need a finer granularity or a bigger domain we have to retrain completely. Finally, the real main issue is that we brought back the mesh.