PINNs for Transport Equation

mouhcine.assouli

May 2021

1 Introduction

The transport equation is a first-order linear partial differential equation (PDE) models the movement of mass from one place to another.
The constraints on this are that the flow is perfectly and instantaneously mixed in the radial direction, which is perpendicular to the direction of momentum. This occurs in instances such as a pipe where the ratio of the radius to the length of the pipe approaches a small number; and so can be modeled as an infinitesimal element applying the tools of calculus. Such instances are common in science, such as in hydrology where a contaminant can enter a stream, in environmental engineering, and in analytic chemistry where the equation can be used to model machine mechanisms such as in the capillary column. I have already encountered a great difficulty in solving this equation with the classic finite elements method [1] and isogeometry finite elements method [2]. The difficulty is in the approximation of the discontinuous functions (shock), in this case, we obtains oscillations of the numerical solution at the points of discontinuity. In this chapter, we develop a PINN-based algorithm for solving the transport equation. Based on a loss function defined by the underlying PDE, we train a DNN to yield the solution of the transport equation. We begin by describing the theoretical underpinnings of the algorithm. After presenting the transport equation and the PINNs deep learning method, we briefly discuss important concepts including the approximation property of neural networks and automatic differentiation. Tying these concepts together, we then show how PINNs can be used to solve the transport equation. Next, we present numerical experiments.

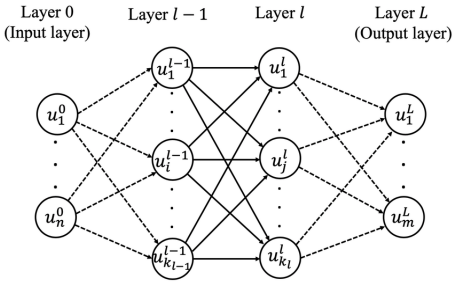

Figure 1: Schematic representation of a feed-forward neural network with L 1 hidden layers.

2 Theory

2.1 Transport Equation

The transport equation is a linear, first-order, hyperbolic PDE of the form:

$$\begin{cases} u_t + cu_x &= 0 \quad in \quad \Omega \times [0,T], \\ u(t=0,x) &= u_0(x) \quad given, \\ u &= 0 \quad on \quad \partial\Omega. \end{cases} \tag{1}$$

where Ω is a domain in R^d with d as the space dimension and T is the final time and c is the velocity defined on Ω, for reason of simplicity we consider $d = 1$ and c constant. On the other hand u_0 it is a function which presents the initial condition.
The exact solution obtained by the method of the characteristics is given by:

$$u(t,x) = u_0(x - ct).$$

2.2 The PINNs deep learning method

Neural network is a set of neurons organized in layers in which evaluations are performed sequentially through the layers. It can be seen as a computational graph having an input layer, an output layer, and an arbitrary number of hidden layers. In a fully connected neural network, neurons in adjacent layers are connected with each other but neurons within a single layer share no connection.
Thanks to the universal approximation theorem, a neural network with n neurons in the input layer and m neurons in the output layer can be used to represent a function $u : R^n \to R^m$ as shown in Figure 1. For illustration, we consider a network of L + 1 layers starting with input layer 0, the output layer L, and L 1 hidden layers. The number of neurons in each layer is denoted as $k_0 = n, k_1, ..., k_L = m$. Each connection between the i-th neuron in layer l 1 and j-th neuron in layer l has a weight $w_j i^l$ associated with it. Moreover, for each neuron in layer l, we have an associated bias term b_i, $i = 1, , k_l = m$. Each neuron represents a mathematical operation, whereby it takes a weighted sum

of its inputs plus a bias term and passes it through an activation function. The output from the k-th neuron in layer l is given as :

$$u_k^l = \sigma\left(\sum_{j=1}^{k_l-1} w_{kj}^l u_j^{l-1} + b_k^l\right) \tag{2}$$

where σ represents the activation function. Commonly used activation functions are the logistic sigmoid, the hyperbolic tangent, and the rectified linear unit. By dropping the subscripts, we can write equation 1 compactly in the vectorial form:

$$u^l = \sigma(W^l u^{l-1} + b^l) \tag{3}$$

where W^l is the matrix of weights corresponding to connections between layers l 1 and l, u^l and b^l are vectors given by u_k^l and b_k^l, respectively, and the activation function is applied element-wise. Computational frameworks, such as Pytorch, can be used to efficiently evaluate data flow graphs like the one given in equation 2 efficiently using parallel execution. The input values can be defined as tensors (multi-dimensional arrays) and the computation of the outputs is vectorized and distributed across the available computational resources for efficient evaluation.

2.3 Approximation property of neural networks

Neural networks are well-known for their strong representational power. It has been shown that a neural network with a single hidden layer and a finite number of neurons can be used to represent any bounded continuous function to any desired accuracy. This is also known as the universal approximation theorem. It was later shown that by using a non-linear activation function and a deep network, the total number of neurons can be significantly reduced. Therefore, we seek a trained deep neural network (DNN) that could represent the mapping between the input (x,t) and the output $u(x,t)$ of the transport equation . It is worth noting that while neural networks are, in theory, capable of representing very complex functions compactly, finding the actual parameters (weights and biases) needed to solve a given PDE can be quite challenging.

2.4 Automatic differentiation

Solving a PDE using PINNs requires derivatives of the network's output with respect to the input. There are four possible ways to compute derivatives : (1) hand-coded analytical derivatives, (2) symbolic differentiation, (3) numerical approximation such as finite-difference, and (4) automatic differentiation (AD). Automatic differentiation, also called algorithmic differentiation, computational differentiation, auto-differentiation, or simply autodiff, is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log,

sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program. However, an efficient implementation of AD can be non-trivial. Fortunately, many existing computational frameworks such as Tensorflow and PyTorch have made available efficiently implemented AD libraries. In fact, in deep learning, backpropagation, a generalized technique of AD, has been the mainstay for training neural networks. To understand how AD works, consider a simple fully-connected neural network with two inputs (x_1, x_2), one output (y), and one neuron in the hidden layer. Let us assume the network's weights and biases are assigned such that:

Forward pass	Reverse pass
$x_1 = 1$ $x_2 = -1$	$\frac{\partial y}{\partial y} = 1$
$\mu = 2x_1 + 3x_2 - 1 = -2$ $h = \frac{1}{1+e^{-\mu}}$	$\frac{\partial y}{\partial h} = \frac{\partial 5h+2}{\partial h} = 5$ $\frac{\partial y}{\partial \mu} = \frac{\partial y}{\partial h}.\frac{\partial h}{\partial \mu} = 0.525$
$y = 5h + 2$	$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial \mu}.\frac{\partial \mu}{\partial x_1} = \frac{\partial y}{\partial \mu} \times 2 = 1.050$ $\frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial \mu}.\frac{\partial \mu}{\partial x_2} = \frac{\partial y}{\partial \mu} \times 3 = 1.575$

Table 1: Example of forward and reverse pass computations needed by AD to compute partial derivates of the output with respect to the inputs at $(x_1, x_2 = (1, -1)$

$$\mu = 2x_1 + 3x_2 - 1$$
$$h = \sigma(\mu) = \frac{1}{1 + e^{\mu}}$$
$$y = 5h + 2$$

where h represents the output from the neuron in the hidden layer computed by applying the sigmoid function σ on the weighted sum of the inputs μ.
To illustrate the idea, let us say we are interested in computing partial derivatives $\frac{\partial y}{x_1}$ and $\frac{\partial y}{x_2}$ at $(x_1, x_2 = (1, -1)$, AD requires one forward pass and backward pass through the network to compute these derivatives as detailed in Table 1.

2.5 Solving the transport equation

To solve the transport equation, we leverage the capabilities of neural networks as function approximators and define a loss function that minimizes the residual of the transport equation at a chosen set of training (collocation) points. This is achieved with (i) a DNN approximation of the unknown $u(x, t)$; (ii) a loss function incorporating the transport equation and sampled on a collocation grid; (iii) a differentiation algorithm, i.e. AD in this case, to evaluate partial derivatives of $u(x, t)$.
Let us define $f(t, x)$ to be given by: $f = u_t + cu_x$. The loss function can now be constructed using a mean-squared-error (MSE) norm as:

$$MSE = MSE_u + MSE_f, \tag{4}$$

where

$$MSE_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|$$

and

$$MSE_f = \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|$$

Here, $\{t_u^i, x_u^i, u^i\}_{i=1}^{N_u}$ denote the initial and boundary training data on $u(t,x)$ and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ specify the collocations points for $f(t,x)$.

3 Numerical experiments.

In this section, we present numerical experiments for the approximation of solutions of the transport equation pde1 by PINNs.
For the numerical experiment, we consider the transport equation in one space dimension i.e, d = 1, domain $\Omega = [5,5]$, T = 1, c=1, and initial data:

$$u_0(x)\begin{cases} 2 & si \quad -1 \leq x \leq 1, \\ 0 & sinon. \end{cases} \tag{5}$$

Then, the exact solution of the transport equation is given by:

$$u(t,x) = u_0(x-t).$$

To get the results, we run algorithm [3] with random training set, $N_u = 200$ (the initial and boundary data) and $N_f = 10000$ (the interior training points), we consider a fully connected neural network with the tanh activation function, with 8 hidden layers and 20 neurons in each layer. Moreover, we use the loss function 3, and the optimizer is the second-order LBFGS method. The results for this procedure are shown in figure (1.2). We see from this figure a better approximation with the relative L_2 error for this case is $9.639037.10^{-2}$. We begin the tests by studying the predictive accuracy of the PINN transport solver for different network architectures. For the considered point-source, we compute solutions by using a number of different architectures formed by varying the number of hidden layers and the number of neurons in each hidden layer. For each architecture, we use $N_u = 200$ (the initial and boundary data) and $N_f = 10000$ (the interior training points). As shown in Table 1,2, we compute relative L_2 error between the predicted solution and the analytical solution, we observe that as we increase the number of layers and neurons, the predictive accuracy of the network is improved as it can approximate more complex functions.
Now we fix the layers and the number of neurons where we find a better approximation from the table 2 i.e $Layers = 6$, $Neurons = 15$, and we compute solutions by varying the number of the initial and boundary data N_u and the interior training points N_f. As shown in Table 3 we observe the same remark as of the first table

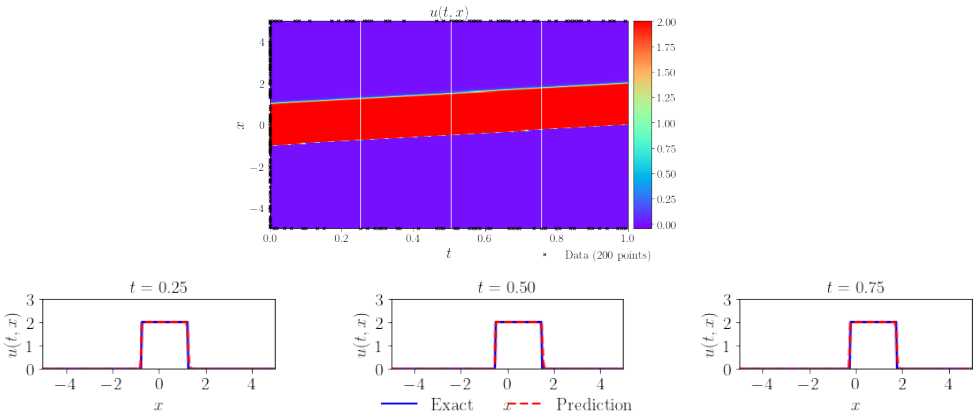

Figure 2: Comparison of the predicted and exact solutions for transport equation.

Layers/Neurons	10	15	20	25
4	$1.900822e-01$	$1.674971e-01$	$1.252714e-01$	$1.690296e-01$
6	$1.284218e-01$	$9.349769e-02$	$1.444393e-01$	$1.412934e-01$
8	$1.230510e-01$	$1.407010e-01$	$1.141542e-01$	$1.348473e-01$
10	$2.880418e-01$	$1.979448e-01$	$3.175755e-01$	$1.251093e-01$

Table 2: Relative L_2 errors between the predicted and the analytical solutions for different number of hidden layers and neurons per layer.

N_u/N_f	1000	5000	10000	50000
100	$2.079933e-01$	$1.930257e-01$	$4.371226e-01$	$2.588320e-01$
150	$2.127333e-01$	$2.105516e-01$	$2.972924e-01$	$2.049043e-01$
200	$2.759039e-01$	$1.042269e-01$	$2.343908e-01$	$1.045913e-01$
250	$1.567476e-01$	$2.063259e-01$	$7.685480e-02$	$1.320272e-01$

Table 3: Relative L_2 errors between the predicted and the analytical solutions for different number of N_u, N_f.

4 Conclusion

As we had seen, the neuron network method is stronger for problems which are complex to solve mathematically.

References

[1] https://colab.research.google.com/drive/1iKBL02ajuk4Hp3nijPYly2TSYfmjE10C?usp=sharing.

[2] https://github.com/Mouhcine-56/S-FEM-S.

[3] https://github.com/Mouhcine-56/Transport_Equ_With_NN

[4] Maaziar Raissi. Deep hidden physics models:deep learning of nonlinear partial differential equations. Journal of Machine Learning Research, 2018.

[5] Maaziar Raissi, Paris Perdikari, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differ- ential equations. Journal of Computational Physics, 2019.