

---

# dNNsolve: an efficient NN-based PDE solver

---

**V. Guidetti**

DESY

Hamburg, Germany

veronica.guidetti@desy.de

**F. Muia**

DAMTP and CTC

University of Cambridge, UK

fm538@cam.ac.uk

**Y. Welling**

DESY

Hamburg, Germany

yvette.welling@desy.de

**A. Westphal**

DESY

Hamburg, Germany

alexander.westphal@desy.de

## Abstract

Neural Networks (NNs) can be used to solve Ordinary and Partial Differential Equations (ODEs and PDEs) by redefining the question as an optimization problem. The objective function to be optimized is the sum of the squares of the PDE to be solved and of the initial/boundary conditions. A feed forward NN is trained to minimise this loss function evaluated on a set of collocation points sampled from the domain where the problem is defined. A compact and smooth solution, that only depends on the weights of the trained NN, is then obtained. This approach is often referred to as ‘PINN’, from *Physics Informed Neural Network* [1, 2]. Despite the success of the PINN approach in solving various classes of PDEs, an implementation of this idea that is capable of solving a large class of ODEs and PDEs with good accuracy and without the need to finely tune the hyperparameters of the network, is not available yet. In this paper, we introduce a new implementation of this concept - called **dNNsolve** - that makes use of *dual Neural Networks* to *solve* ODEs/PDEs. These include: i) sine and sigmoidal activation functions, that provide a more efficient basis to capture both secular and periodic patterns in the solutions; ii) a newly designed architecture, that makes it easy for the the NN to approximate the solution using the basis functions mentioned above. We show that **dNNsolve** is capable of solving a broad range of ODEs/PDEs in 1, 2 and 3 spacetime dimensions, without the need of hyperparameter fine-tuning.

## 1 Introduction

The modern era of the physical description of nature is built upon a very fundamental realization starting with Galilei’s and Newton’s work – namely that the dynamical laws of nature are given in general by systems of coupled Ordinary and Partial Differential Equations (ODEs and PDEs), that is the laws are *local* in space-time. Since then, it is also clear that describing the state of a physical system and/or predicting its time evolution into the future requires specifying appropriate boundary and/or initial conditions, and then solving the PDEs on the relevant spatial or space-time domain.

Attempting to do so shows very quickly, that finding exact analytical solutions for any situation involving more than a few degrees of freedom of the system in question becomes prohibitively difficult in most cases. Therefore, solving PDEs for most problems became critically dependent on developing approximation schemes relying on discretizing space-time and converting the PDEs in finite-difference equations, which can then be solved *algorithmically* in an iterating series of

steps. This process eventually led to the development of finite-difference and finite-element method (FD/FEM) based discretized integrators. These discrete methods of integrating ODEs and PDEs were developed into standardized software packages being able to produce numerical adaptively discretized solutions to ODEs and PDEs with controllable approximation errors for many types of ODEs and PDEs from physics and mathematics. From hereon, we will use FD/FEM to summarily denote both the PDE solvers as well as the ODE solvers.

Finite difference methods are clearly more straightforward to be implemented. This is the main reason why they are widespread in the scientific community, see for instance [3–12] for some examples of early universe, general relativity and large scale structure simulations. On the other hand finite difference methods do not work well if the domain is irregular, which is a standard situation in many engineering contexts, in which one has to deal for instance with structural analysis of buildings or deformation and stresses of solid bodies. There is a number of finite element based softwares available on the market, such as [13–15] and a few open source ones [16–19]. Both finite element and finite difference methods achieve outstanding results in terms of accuracy, but share a few shortcomings: first, the discretization is unavoidably a source of error. This problem can be cured in most situations by improving the precision of the solver, i.e. increasing the number of points in the grid for the finite difference methods and refining the mesh and/or choosing a basis of higher order polynomials to approximate the solution in the finite element case. Of course, this comes at the cost of larger simulation times. However, the second and probably most important problem of these methods is that they are very memory expensive, as the amount of information to be stored at any time is proportional to the number of points in the grid for finite difference methods, and proportional to the number of elements of the mesh for finite element methods. Sometimes, this comes at the expense of not being able to use faster processors, e.g. GPUs instead of CPUs, to perform large simulations.

More fundamentally, the approximation quality of FEM integrators depends on the cleverness of the chosen higher-order discretization schemes of the solving function and its space-time derivatives entering the given ODE/PDE as well as the choice of the interpolating functions used to improve the discretized integration method. While they have become quite good, they were developed by humans based on certain criteria and motivations, which prevent these methods from being adapted from the ground up to the specific ODE/PDE in question. Instead, the methods developed for the FD/FEM paradigm have a given fixed algorithmic form but are set up to deliver reasonably accurate solutions to many ODEs/PDEs.

Thus, it is conceivable to achieve a potentially significant improvement over the FD/FEM solvers, if the functional form of the approximation algorithm itself could become dynamically adaptable to each given ODE/PDE at hand. For human programmers this would imply to re-invent a new FD/FEM algorithm each time – hence it is here where the specific properties of machine learning based on deep neural networks (DNNs) become useful. Based on the universal approximation theorem [20–22], a DNN can act as a universal regressor which is able to represent any bounded continuous function, given suitable discretized training data, without pre-specifying a human-developed symbolic algorithm discretizing the function.

## 1.1 Relevant literature

The interplay between neural networks and differential equations presumably started back in the early '90s, when an algorithm based on the Hopfield network combined with finite difference methods was proposed in [23]. The idea to use a deep neural networks was then put forward in [24–26] in the second half of the '90s. The idea of these papers was very simple: a neural network can be used to approximate a function as well as its derivatives, and it can be trained in order to minimize a loss function vanishing when the PDE is solved. At that time, backpropagation was not widespread yet, hence the authors had to restrict to a quite simple architecture with a single hidden layer, in order to be able to compute derivatives analytically.

From here it is short step to the PINN, introduced in 2017 by [1, 2] – the physics-informed neural network. The idea is simple – you use the learning and approximating power of the DNN to adjust the learned function until it satisfies a given ODE or PDE precisely enough. This is achieved by using the square of the differential equation itself, evaluated on the DNN output (its best guess of the function at a given algorithmic step called ‘epoch’), as the ‘bulk’ loss function whose gradients then tell the DNN how to adjust its weights using backpropagation. The full loss function here contains separate pieces which enforce a solution of the PDE in the bulk space-time as well as fulfillment of

the initial and boundary conditions determining the solution. Such a PINN will in principle learn an optimized approximate solution of the ODE/PDE given to it, instead of using a fixed symbolically prescribed ‘one-size-fits-all’ algorithm.

This may lead to gains in the approximation quality and/or efficiency of the dynamically learned algorithm compared to pre-fixed algorithm, and has the potential to become highly memory-conserving. A PINN, once trained to solve a given ODE/PDE only needs to store the architecture of its DNN and its trained weight system (one set of float valued matrices). On the other hand, a conventional solver has to build the discretized function system and its derived higher-order interpolating functions and, moreover, needs to maintain this information in memory during work

An interesting alternative approach to PINNs, named the CINT (from Constrained Integration) algorithm, was proposed in 2014 in [27, 28], based on a combination of the CPROP algorithm [29] with Galerkin methods [30–32]. The CINT algorithm was shown to successfully solve the wave and heat equation, and achieves very good results both in terms of speed and accuracy.

We now return to a closer discussion of the PINN. As already mentioned, the loss function of a PINN contains several pieces, which separately enforce the PDE solution in the bulk and its initial and boundary conditions. Using rather large networks (such as for instance a 9-layers NN containing 20 units for each layers for the Burgers’ equation, for a total of more than 3000 parameters), the authors of [1] achieved quite good results, reaching a loss  $< 10^{-3}$  for a variety of PDEs, such as the Burgers’ and the Schrödinger equations in 1 + 1D.

A more sophisticated architecture was proposed under the name of *Deep Galerkin Method* (DGM) in 2018 in [33] with the aim of solving high-dimensional PDEs. The architecture is reminiscent of LSTM models, and is designed to successfully detect sharp turns and in the solution. The DGM architecture makes use of distributed training and a Monte-Carlo method for the computation of second derivatives to solve various high-dimensional (up to 200 dimensions) PDEs such as the Burgers’ equation and the Hamilton-Jacobi-Bellman equation.

More recently, various packages exploiting these ideas have been put forward. For instance, *DeepXDE* introduces a residual-based adaptive refinement of the grid in order to improve the training efficiency of PINNs [34]. DeepXDE is shown to successfully solve PDEs that are first order in time derivative as well as integro-differential equations and inverse problems. Very nicely, the DeepXDE Python library can be easily adapted to work on non-trivial domain shapes. On the other hand, *Pydens* [35] is a Python library that allows to solve a large family of PDEs including the wave equation and the heat equation, and gives the possibility to easily switch between standard architectures (such as ResNet and DenseNet).

Beyond the ideas mentioned above, there are many more that would need to be mentioned, including: the use of adaptive activation functions to accelerate the convergence of PINNs [36, 37], the use of convolutional neural networks to solve PDEs or to address the inverse problem [38–40], adversarial training techniques [41], energy based approaches [42], the use of random approximation methods [43–46]. Note that it has been estimated that techniques based on PINNs might become competitive with standard methods such as finite difference [47]. See also [48] for a recent discussion of the advantages and disadvantages of using neural network based techniques for solving PDEs and about the possible routes forward in this direction.

## 1.2 Our contribution

At the moment though, we are not able to find a single implementation of the PINN or any of the other ideas that is capable of solving a large class of ODEs and PDEs with good accuracy and without the need to finely tune the hyperparameters of the network. By ‘large class of ODEs and PDEs’ we mean to include not only the simplest, mandatory examples, such as the wave equation, the heat equation and the Poisson equation, but also more complex examples that include for instance damped oscillations and stiff equations.

The crucial idea we report here, which makes the PINN more efficient to use for solving many different ODEs/PDEs, is to develop a DNN architecture which optimizes the set of functions used in the DNN’s exploitation of the universal function theorem. Namely, we first observe that bounded solutions of any ODE/PDE can in principle be decomposed into any given complete orthonormal

function system. Fourier decomposition and the softened-bin decomposition of a function, used to prove the universal approximation theorem for feed-forward DNNs, are two examples.

In our experience in physics, the various types of dynamics shown by ODE and PDE solutions, the resulting function very often is composed out of a secular non-oscillatory part, and a part which is a Fourier synthesis of various oscillatory components. While it is possible to approximate a secular slow-changing function solely by Fourier synthesis, doing so requires summing over a very large number of Fourier modes to achieve satisfactory accuracy. Conversely, approximating an oscillatory function over many periods using the softened-bin decomposition requires summing up a very large number of narrow bins. Both decompositions are possible, although inefficient, for the each of two the respective cases.

Hence, for achieving a given approximation quality, we expect it to be much more efficient, to use both the secular behaviour of the function system used to build the softened-bin decomposition and Fourier decomposition – the first to efficiently approximate the secular components of a function, the second to model its oscillatory components. Such a ‘two-stream’ decomposition will likely require far fewer terms/modes to approximate a function to given finite accuracy, than each of the two decomposition streams by itself. For a similar approach in the context of time-series analyses, see for instance [49].

Therefore, we propose **dNNsolve**: the ‘two-stream’ decomposition into the DNN of the PINN by splitting some of the hidden layers into two parts whose choice of activation function is suitable for the non-oscillatory secular decomposition in one part of the layer, and for a Fourier decomposition in the second part of the layer (using periodic activation functions that were previously used for instance in the context of Fourier NNs [50, 51]). After passing this ‘two-stream’ layer, the two decomposition streams are algebraically combined to produce the DNN output.

We test **dNNsolve** on a broad range of ODEs and PDEs in 1, 2 and 3 dimensions. The main improvements on the one-stream PINNs are:

- We use the *same* hyperparameters (such as learning rate, network size and number of points in the domain) for each  $d$ -dimensional PDE and achieve good accuracies for all of them.
- A small NN architecture with  $d \times \mathcal{O}(100 - 200)$  trainable parameters is effective for the problems at hand, while keeping the number of sampled points at  $\mathcal{O}(1000)$ . On the other hand, in e.g. [34] PINNs with  $\mathcal{O}(1000 - 8000)$  parameters were employed for 1- and 2-dimensional examples.
- We use only  $\mathcal{O}(1000)$  epochs of ADAM (or  $\mathcal{O}(100)$  with mini-batches) followed by a variable number of iterations of BFGS until convergence. This may be contrasted with the 15000 – 80000 epochs of ADAM used in [34].

The plan of this note is as follows. In section 2 we will proceed to describe the setup of the two-stream PINN, and discuss its loss function as well its weight initialization and hyperparameter choices. In section 3, we present its application to significant number of ODEs and PDEs and (0+1), (1+1) and (2+1) dimensions. To gauge the accuracy of the solutions learned, we compare an accuracy estimate based on the root mean squared (RMS) error provided by the PINN loss function as well as the RMS error between the PINN solution and the known analytical solution for each example. A lot of the technical details are relegated to several appendices A, B, C, D.

## 2 Dual base network

The aim of our work is to find an improved PINN architecture that is able to solve a general differential equation [26]

$$G(\vec{x}, u^l(\vec{x}), \nabla u^l(\vec{x}), u^l(\vec{x}), \nabla^2 u^l(\vec{x}), u^l(\vec{x}), \dots) = 0, \quad \vec{x} \in \mathcal{D} \subset \mathbb{R}^D, \quad \vec{u}(\vec{x}) \in \mathbb{R}^{n_o}, \quad (1)$$

subject to certain boundary and initial conditions, where  $\mathcal{D}$  is the domain,  $D$  is the number of spacetime dimensions and  $n_o$  is the number of components of  $\vec{u}$ . Therefore we look for a scalable architecture that can be easily adapted to problems having different input and output domain dimensions. In order to do this, we take direct inspiration from series expansion. The universal approximation theorem in its ‘arbitrary width’ formulation [20–22] states that a feed forward NN with a single hidden layer and arbitrary continuous activation functions can approximate every continuous functions with

arbitrary precision. On the other hand, carefully choosing the architecture and activation functions may be useful for computational purposes, reducing the amount of neurons needed to get the solution in a finite amount of time and with finite computational power.

One of the best-known tools to perform function decomposition is given by the Fourier series that allows one to decompose each piece-wise continuous function on some given interval. Nevertheless, also in this case, an exact function decomposition may be given by an infinite sum of harmonic terms (a simple example is given by the infinite series representing a straight line). Trying to reduce the number of nodes required to obtain a satisfactory approximation of our PDE solution, given that we can not know a priori its behaviour, we decide to use two kinds of activation functions, i.e. sine and sigmoid functions. The latter can be used to reduce the number of neurons used in capturing the secular variations of the solution, while the sine performs a Fourier-like decomposition. We combine the two types of activation functions both linearly and non-linearly, trying to improve the robustness and flexibility of our network. Our network architecture, dubbed `dNNsolve` is given by a feed forward NN with multiple inputs and outputs, five hidden layers (of which only two layers contain trainable parameters) and two branches representing the periodic and non-periodic expansion of the solution. In the first layer each input variable is processed independently by two dense layers having the same number of neurons  $N$  with sigmoidal and sinusoidal activation function, respectively:

$$\begin{array}{ccc} & x^i & \\ & \swarrow & \searrow \\ \sin(\omega_k^{(i)} x^i + \phi_k) \equiv f_k^{(i)} & & \sigma(w_k^{(i)} x^i + b_k) \equiv s_k^{(i)} \end{array}$$

where  $k = 1, \dots, N$  and  $i = 1, \dots, D$ , where  $D$  is the number of input variables, i.e. the number of spacetime dimensions of the problem. Afterwards, the periodic and non-periodic outputs are merged together using three multiplication layers, see Figure 1. These perform the element-wise multiplication of the neurons belonging to different input variables. This operation is performed for the two branches separately and for their product to take into account some non-linear interplay between the periodic and non-periodic decomposition of the solution. The output of the multiplication layers, identified by  $\mathbf{F}$ ,  $\mathbf{S}$  and  $\mathbf{FS}$ , can be summarized as:

$$\mathbf{F} = \{F_k\} = \left\{ \prod_{j=1}^D f_k^{(j)} \right\}, \quad \mathbf{S} = \{S_k\} = \left\{ \prod_{j=1}^D s_k^{(j)} \right\}, \quad \mathbf{FS} = \{F_k S_k\}, \quad (2)$$

where  $k = 1, \dots, N$ . These results are then joined together using a concatenation layer that is taken as input by a dense layer,  $\mathbf{D}$ , with no activation function having  $n_o$  neurons, where  $n_o$  is the number of unknown functions we need to compute as output of our network (this depends on the problem at hand):  $\mathbf{D}_l(y) = d_l y^i + a_l$ ,  $l = 1, \dots, n_o$ . In the case of a single ODE with scalar input  $x$ , the output of our network is a single function  $\hat{u}$ . In this case,  $D = 1$  and  $n_o = 1$ , and the output can be written as:

$$\hat{u}(x) = \sum_{k=1}^N d_k \sin(\omega_k x + \phi_k) + d_{N+k} \sigma(w_k x + b_k) + d_{2N+k} \sin(\omega_k x + \phi_k) \sigma(w_k x + b_k) + a, \quad (3)$$

where we have denoted  $d_{k1} \equiv d_k$  and  $a_1 \equiv a$ . A pictorial representation of our architecture for multiple inputs can be found in Figure 1.

Let us also mention that we tried several alternatives to the sine and sigmoid activation functions. In the secular branch of the network we also tried for instance with several possible activation functions: tanh, Gaussian, Hermite polynomials,  $\tanh(x^p)$ . In the Fourier part of the network we made a few experiments with wavelet layers [52]. Overall, we observed that the combination of sinusoidal activation functions with sigmoids is the best choice both in terms of accuracy and in terms of training efficiency.

**Weight initialization** The number of network parameters is given by  $4ND + n_o(3N + 1)$ , it scales linearly both in the number of input variables  $d$  and in the number of neurons per branch  $N$ . After trying different possibilities, we choose the following weight initialization procedure:  $d_k = 10^{-4}$ ,  $w_k^{(i)} \sim \mathcal{U}(0, 10^{-3})$ ,  $\omega_k^{(i)} \sim \mathcal{U}(\pi/(x_i^{\max} - x_i^{\min}), N\pi/(x_i^{\max} - x_i^{\min}))$  where  $N$  is the number of

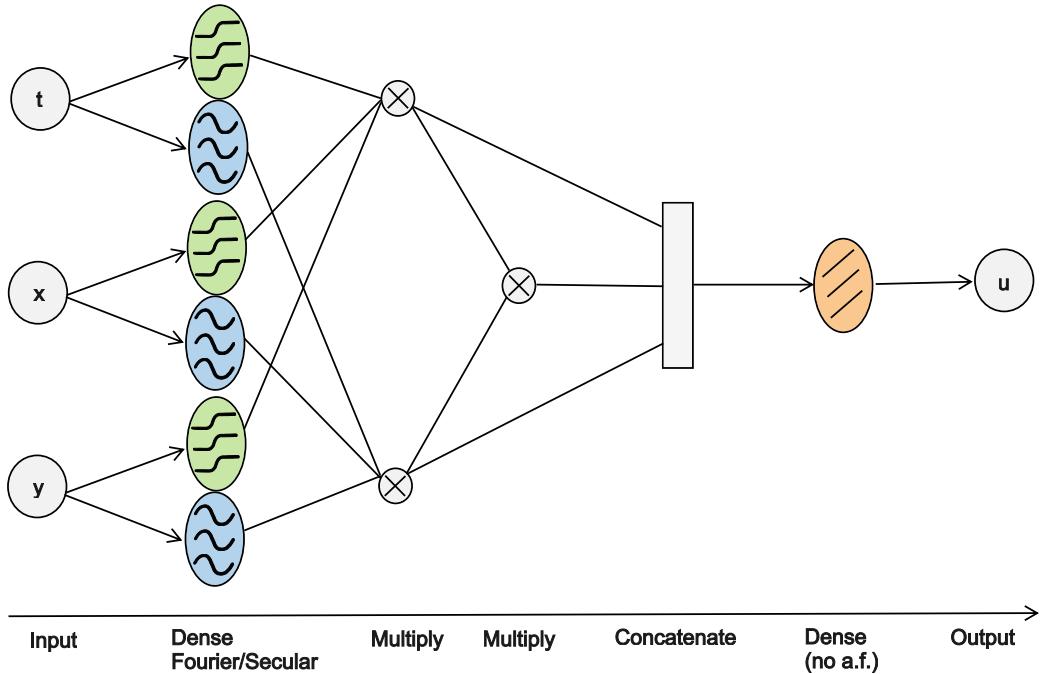


Figure 1: **dNNsolve** network architecture for the case  $D = 3$ : multiple input variables are processed independently using Dense layers with sine and sigmoid activation functions. The results are then processed by three multiplication layers and finally merged together using a concatenation layer. The final output (scalar in this picture) is then computed using a simple dense layer with no activation function.

neurons per branch. Biases,  $a_l$ ,  $\phi_k$ ,  $b_k$ , are always initialized to zero. The choice of the initialization for the frequency of the sine activation functions reflects some prior knowledge about the physical scales that characterize the system: for instance in the one-dimensional case, we do not expect hundreds of oscillations in the domain of definition of the ODE. The lowest frequency is chosen to be first harmonic since we want the sine activation functions to capture only the oscillating part of the solution as lower frequencies can be easily captured by sigmoid neurons. Regarding the amplitude parameters  $d_i$ , we choose to initialize them to tiny values, i.e. we start from nearly vanishing solution. This decision is related to the behaviour of the bulk contribution to the loss function. In case the ODE/PDE is homogeneous and admits a zero solution, in the first steps of training the algorithm tends to drive  $d_i$  to zero before it starts learning the real solution. This happens because  $\hat{u} = 0$  appears to be a local minimum of the loss function. The amplitude initialization we choose is then able to shorten training under certain conditions that we describe in the next section. Our choice of processing input variables independently, instead of using them as a single input for a dense layer, is motivated by empirical observation. Dense layers with sinusoidal activation function over linear combinations of inputs seemed to be less flexible during training, i.e. the system got systematically stuck in local minima. In order to overcome this problem, fine-tuned frequency weights initialization was required, leading to a poor level of generalization of the network. As opposed to sigmoid activation functions, sinusoidal functions in 2/3D are non localised in space and do not adapt well: a small change in one parameter implies changing the function in the whole area/volume. This makes their training very hard since the loss function turns out to be fulfilled with local minima [53, 54]. We empirically found that this behaviour can be mitigated by processing input variables independently. This may be due to the presence of extra phase parameters ( $\phi_k$ ) or because we reduce the effects of changing weights to a single dimension.

**Loss function** Following [33] we use a piece-wise loss function, given by the sum of different contributions coming from bulk, boundary and initial conditions points. Let us write the domain of our problem splitting it into time and space components:  $\mathcal{D} = [t_0, t_L] \times \Omega$  where  $\Omega \subset \mathbb{R}^{D-1}$ . Different PDEs, see Eq. (1), having different degrees in time and space variables can be subject to different conditions. In the examples showed in this work we mainly considered

- Initial conditions:

$$u^l(t_0, \vec{x}) = u_0^l(\vec{x}), \quad [\partial_t u^l](t_0, \vec{x}) = v_0^l(\vec{x}) \quad (4)$$

- Boundary conditions:

$$u^l(t, \vec{x})|_{\partial\Omega} = f^l(\vec{x}) \quad (\text{Dirichlet}), \quad (5)$$

$$[\partial_{\vec{n}} u^l(t, \vec{x})]|_{\partial\Omega} = g^l(\vec{x}) \quad (\text{Neumann}), \quad (6)$$

where  $\vec{n}$  is the normal to the boundary unit vector.

Collectively denoting by  $\theta = \{w, \omega, \phi, b, d, a\}$  the set of NN parameters and  $\hat{u}^l(\vec{x}, t, \theta)$  the output of the NN, the general shape of the loss function is given by:

$$\begin{aligned} \mathcal{L}(t, \vec{x}, \theta, \alpha) &\equiv \mathcal{L}_\Omega + \mathcal{L}_0 + \mathcal{L}_{\partial\Omega} = \\ &= \sqrt{\frac{1}{n_\Omega} \sum_i [G(t_i, \vec{x}_i, \hat{u}^l(t_i, \vec{x}_i, \theta), \nabla \hat{u}^l(t_i, \vec{x}_i, \theta), \dots)]^2} + \\ &+ \alpha_0 \sqrt{\frac{1}{2n_0} \sum_j ([\hat{u}^l(t_0, \vec{x}_j, \theta) - u_0^l(\vec{x}_j)]^2 + [[\partial_t \hat{u}^l](t_0, \vec{x}_j, \theta) - v_0^l(\vec{x}_j)]^2)} + \\ &+ \alpha_{\partial\Omega} \sqrt{\frac{1}{n_{\partial\Omega}} \sum_k [u^l(t_k, \vec{x}_k, \theta)|_{\partial\Omega} - f^l(t_k, \vec{x}_k)]^2}, \end{aligned} \quad (7)$$

where we considered Dirichlet boundary conditions and second order PDEs in time. We take the sum of square roots instead of the sum of squares, because empirically we observe that this typically converges faster and is more accurate. If the boundary condition is not Dirichlet, the last term of the loss should be modified accordingly. Also, if the PDE is first order in time, the second term in the root of the second line of Eq. (7) disappears. The loss function is computed over a batch of size  $n_s = n_\Omega + n_{\partial\Omega} + n_0$  where  $n_\Omega$ ,  $n_{\partial\Omega}$  and  $n_0$  are the number of points belonging to the bulk, the boundary and the initial condition respectively. Our loss function depends on two parameters,  $\alpha_0$  and  $\alpha_{\partial\Omega}$  that can be considered as additional hyperparameters of the neural network. This kind of approach was also used in other works as [34, 55]. These parameters are needed to let the NN be able to get out of local minima. For example, consider the case where an homogeneous PDE contains an identically zero solution and boundary conditions set the function to be vanishing on  $\partial\Omega$ . If we consider  $\alpha_0 = \alpha_{\partial\Omega} = 1$  we have that 2/3 of the loss function drive the solution toward the local minimum  $u \equiv 0$ . From empirical observations we saw that the extent of this problem increases with the number of input dimensions  $d$ . On the other hand, taking  $\alpha_0 \gg 1$ ,  $\alpha_{\partial\Omega}$  will introduce a hierarchy in  $\mathcal{L}$ , pushing the solution away from zero (see [56] for a work in an analogous direction). These weights can in principle also be used to put some priority in the piece-wise loss function: if all weights are equal and different parts of the loss function do not share local minima or saddle points, the convergence rate is slowed down. Some intuitions about the relation between different pieces of the loss function and the role played by  $\alpha$  hyperparameters can be found in App. A.

**Optimizer and training** Loss functions often exhibit pathological curvature, saddle points and local minima. Some of these problems can be addressed using adaptive SGD algorithms, as ADAM. The use of mini-batches represents a source of stochasticity that helps in getting out of local minima, while momentum is able to get away from saddle points and proceed in pathological curvature regions accumulating the gradient over the past steps to determine the direction to go. Moreover, momentum gives a one-shot chance of getting out of local minima. As explained in the previous paragraph, the problem we aim to solve may suffer from the presence of local minima. Some of these can be due to discretization, others derive from the definition of the loss function, see App. A. For this reason we decide to perform a two-step training using different optimizers. In the first step we use the ADAM [57] optimizer for a fixed amount of epochs using mini-batches of 256 data. We start

from a huge learning rate  $\lambda = 0.1$  and use the standard Tensorflow learning rate decay callback with parameter choices which reduce  $\lambda$  by a factor  $\delta = 0.5$  when the loss does not decrease by an amount of  $10^{-4}$  in 30 iterations. ADAM training should end in a neighborhood of the global minimum. In the second step, we use BFGS algorithm [58, 59], a second order quasi-Newton method that estimates the curvature of the parameter space via an approximation of the Hessian. This is able to speed up the convergence rate, reaching the minimum in a small number of iterations. We use BFGS without using mini-batches so as to have the best approximation of the loss volume. Training ends when the supremum norm of the gradient vector is below  $10^{-8}$ .

**Validation of the results** In order to be able to check the accuracy of the results, we mainly considered PDEs with analytic solutions, with a few notable exceptions that will be discussed in due course. The accuracy of the solution will be expressed in terms of the root mean squared error  $r$ , that in the case of an initial/boundary value problem is defined as

$$r = \sqrt{\frac{1}{n_{\text{tot}}} \sum_i \left| \hat{u}(t_i, \vec{x}_i, \tilde{\theta}) - \tilde{u}(t_i, \vec{x}_i) \right|^2}, \quad (8)$$

where  $\hat{u}(t, \vec{x}, \tilde{\theta})$  is the NN output evaluated using the trained weights  $\tilde{\theta}$ , while  $\tilde{u}(t, \vec{x})$  is the analytic solution evaluated at the point  $(t, \vec{x})$ . The  $n_{\text{tot}}$  collocation points  $(t_i, \vec{x}_i)$  are taken on a regular grid with  $\{200, 50, 30\}$  points per spacetime dimension in 1D, 2D and 3D respectively. Note that, even for a second order PDE in time, we do not have a separate initial condition and boundary piece, as with  $r$  we only want to measure the average distance from the true solution. In the subsequent sections, we will report the loss values (both the total loss as well as the various components of the loss in Eq. (7)), to illustrate how the training proceeds, and the root mean squared error  $r$ , to evaluate the accuracy of the solutions obtained using `dNNsolve`.

### 3 Results

We study the performance and the robustness of our architecture in solving 1D ODEs and 2D/3D PDEs. In order to do so, we use the same weight initialization in all examples. Moreover, we consider the same number of points and architecture in all equations having same dimensionality, i.e.  $D$  and  $n_o$ . The same criterion is applied to hyper-parameters: we try to keep them fixed within each group of ODEs/PDEs. Numerical results show that this was possible for 1D/2D problems, while 3D problems required to slightly adjust the  $\alpha$  parameters for the reasons described in the previous paragraph.

Please, note that the results that we are listing in this section do not correspond to our best results, as they are obtained using a random set of initialized weights as opposed to the best solution selected out of multiple runs [34].

All the computations were performed with a laptop whose specifics are:

- Machine: MacBook Pro (Retina, 15-inch, Mid 2015),
- Processor: 2.8 GHz Intel Core i7 (quad-core, 64bit, hyperthreading = 8 logical cores),
- Memory: 16 GB 1600 MHz DDR3,
- OS: MacOS Mojave 10.14.6.

We used Tensorflow 2.4 [60, 61]. All the computations were automatically parallelized via hyper-threading over 8 CPU cores. The RAM usage was always less than 500 MB for all the equations in any number of spacetime dimensions.

#### 3.1 1D ODEs

In 1D, we considered several initial value problems on the domain  $t \in [0, 20]$  of the form

$$G(t, u(t), u'(t)) = 0, \quad u(t_0) = u_0, \quad u'(t_0) = v_0, \quad (9)$$

where the constraint on initial first derivative appears only in second order problems. In addition we solve a representative example of boundary value problem of the form

$$G(t, u(t), u'(t)) = 0, \quad u(t_0) = u_0, \quad u(t_L) = u_L. \quad (10)$$

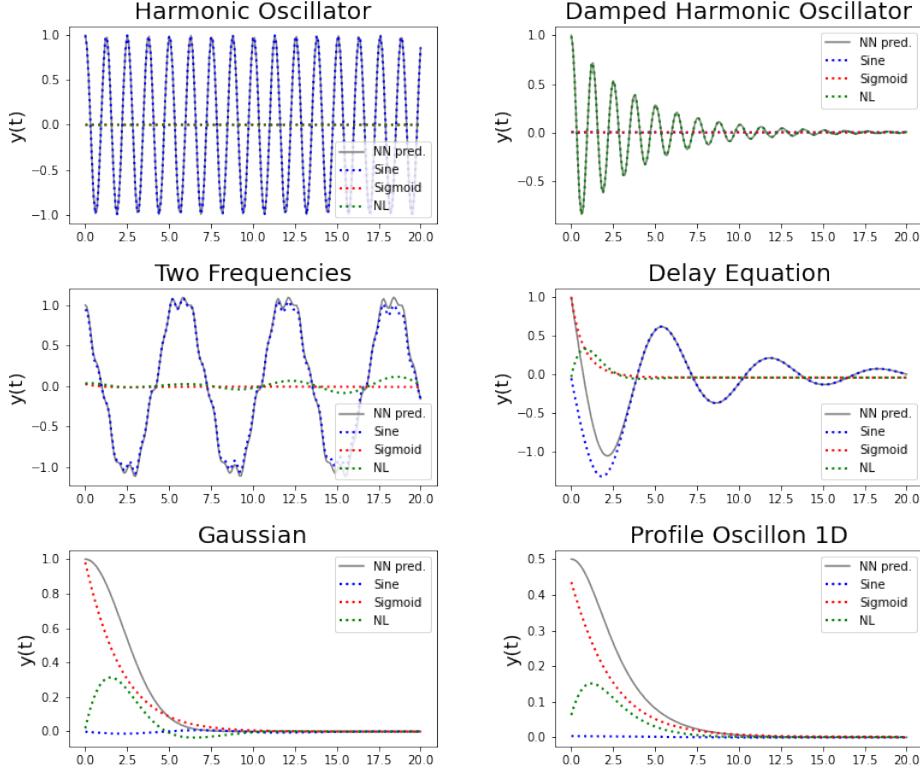


Figure 2: Contributions coming from sine, sigmoid and non-linear branches to the solution of six of the 1D equations considered. As expected, the harmonic oscillator and its damped version are completely solved by the sinusoidal and non-linear branches respectively. In the two frequency example the sinusoidal branch is mainly used, but as it is clear from the solution at  $t > 10$ , noisy contributions from unused neurons are not completely switched off, which calls for a better optimisation. For the Delay Equation the sinusoidal branch captures the oscillatory part of the solution, while the other branches take care of the linear part. For the Gaussian and Oscillon profile equations, the non-linear and secular branches contribute to the final solution, while the sinusoidal is switched off.

as well as a first order delay equation of the form

$$G(t, t', u(t), u(t'), u'(t)) = 0, \quad u(t|t < t_0) = f(t). \quad (11)$$

The explicit ODEs under study are listed in App. B. In all the examples we used  $\{n_\Omega, n_0\} = \{2000, 1\}$  collocation points and  $n = 35$  neurons per branch. The oscillon profile equation, Eq. [10] is a particular case: it is a boundary value problem in 1D and it is typically solved using the *shooting method*. As it is particularly sensitive to the noise of unused neurons, we use only  $n = 10$  neurons per branch in that particular example. It was possible to successfully solve all the equations reported in App. B by fixing  $\alpha_0 = 1$ . The number of epochs using ADAM optimizer is 150, while the total number of epochs, including the ones that use BFGS is reported in Tab. 1.

From our results we can see that the **dNNsolve** is able to learn different kinds of solutions without changing the setting of weight initialization and hyperparameters. The network is able to learn which function basis can better approximate the solution and reduce the contributions coming from the other branches. On the other hand, the capability of really switching off redundant degrees of freedom still needs to be optimized. Six examples showing the interplay between different branches can be found in Figure 2. We see that, despite the network is able to focus on the right basis, it happens that spurious contributions may appear from the other branches as in the ‘two frequencies equation’, Eq. [9]. This problem becomes more apparent when the number of neurons per branch increases.

In Figure 3 we illustrate the evolution of the neural network solutions of the Mathieu equation, see Eq. [1], and the double frequency equation, see Eq. [9], as function of number of epoch. In case

1D results						
			$\log_{10}$			
ODE	Time(s)	Epochs	$\mathcal{L}$	$\mathcal{L}_\Omega$	$\mathcal{L}_0$	$r$
[1]	16.2	1283	-3.6	-3.6	-6.6	-3.6 (*)
[2]	6.3	393	-4.1	-4.1	<-25	-4.1
[3]	10.6	728	-5.0	-5.0	-7.2	-5.8
[4]	11.7	765	-4.6	-4.6	-7.2	-5.1
[5]	6.8	513	-4.2	-4.2	<-25	-3.1
[6]	7.1	414	-3.2	-3.2	<-25	-2.5 (*)
[7]	7.4	642	-3.2	-3.2	<-25	-4.3
[8]	5.5	380	-3.5	-3.5	<-25	-3.7
[9]	10.0	622	-3.9	-3.9	-6.2	-4.2
[10]	7.4	74	-4.9	-4.9	-6.8	-5.2

Table 1: ODE results, entries marked with an asterisk are computed comparing our results with SciPy 1.5.0 `odeint` solver and `ddeint` 0.2 solver with 20.000 points.

of the double frequency equation we note that the neural network learns the low base frequency and the high frequency modulations simultaneously. This has to be contrasted with the expected behaviour of neural network interpolation, which typically learns from low to high frequencies [62]. However, since the loss function of `dNNsolve` is sensitive to the derivatives of the learned function as well, which are more enhanced around the high frequency modulations, the observed behaviour makes sense. The same observation was made in [34], although in their example the corresponding differential equation contained the range of frequencies explicitly as a force term. In case of the Mathieu equation we see that the learned solution remains close to a cosine for quite some training iterations until the BFGS optimizer manages to perturb it enough to quickly converge to a better approximation.

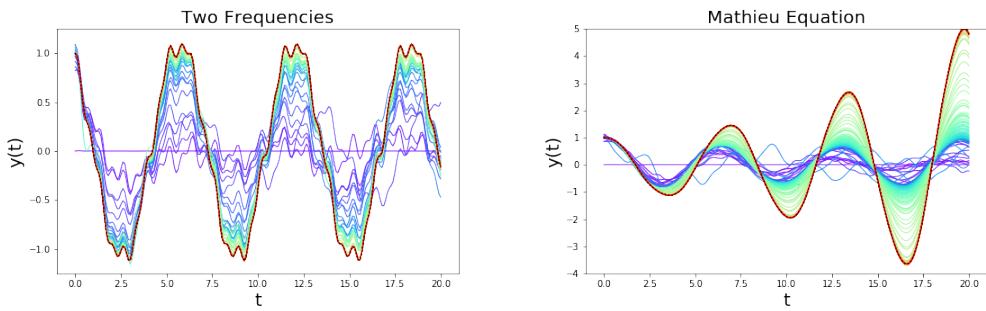


Figure 3: Visualization of the evolution of the solution as represented by the neural network of the double frequency equation, Eq. [9], and the Mathieu equation, Eq. [1], left and right figure respectively. The number of training iterations runs from purple (a few) to red (all of them).

### 3.2 2D PDE

In 2D, we considered several boundary value problems on the domain  $(x, y) \in [0, 1]^2$  with both Dirichlet and Neumann boundary conditions, as well as initial/boundary value problems on the domain  $(t, x) \in [0, 1]^2$ . We refer to App. C for the explicit equations under study. In the 2D case we used  $\{n_\Omega, n_{\partial\Omega}, n_0\} = \{1000, 200, 200\}$  collocation points and  $n = 10$  neurons per branch.

2D results							
			$\log_{10}$				
PDE	Time(s)	Epochs	$\mathcal{L}$	$\mathcal{L}_\Omega$	$\mathcal{L}_0$	$\mathcal{L}_{\partial\Omega}$	$r$
[1]	8.2	528	-4.5	-4.6	-6.4	-5.3	-5.3
[2]	7.5	538	-4.9	-4.6	-6.4	-5.7	-6.3
[3]	5.1	496	-5.1	-5.3	-7.1	-6.1	-6.1
[4]	8.6	911	-3.3	-3.4	-6.8	-5.1	-4.6
[5]	24.8	3764	-2.8	-2.8	-5.0	-4.0	-3.9
[6]	8.1	762	-3.2	-3.2	-6.8	-5.3	-4.5
[7]	7.5	533	-6.4	-5.2	-6.2	-6.3	-6.4
[8]	13.4	1437	-3.5	-3.6	-5.5	-4.4	-4.8
[9]	9.3	942	-4.2	-4.3	-7.0	-4.9	-5.2
[10]	22.9	3744	-2.7	-3.5	-3.9	-3.5	-3.9 (*)
[11]	9.5	1158	-3.6	-3.6	N/A	-5.2	-5.4
[12]	28.7	4574	-3.3	-3.5	N/A	-3.5	-5.6 (*)

Table 2: 2D PDEs results, entries marked with an asterisk are computed comparing our results with Wolfram Mathematica NDSolve results.

Furthermore, it was possible to successfully solve all the equations reported in App. C by fixing the loss weights to  $\{\alpha_0, \alpha_{\partial\Omega}\} = \{10, 1\}$ . The number of epochs using ADAM optimizer is 210, while the total number of epochs, including the ones that use BFGS is reported in Tab. 2.

### 3.3 3D PDE

In the 3D case we considered several boundary value problems in the domain  $(x, y, z) \in [0, 1]^3$ , as well as initial/boundary value problems in the domain  $(t, x, y) \in [0, 1]^3$ . However, we need to distinguish the specifics of the solution case by case. For all the 3D PDEs listed in App. D we used  $\{n_\Omega, n_{\partial\Omega}, n_0\} = \{1000, 1200, 500\}$  collocation points and we considered 210 ADAM epochs. For the vorticity equation and the Lamb-Oseen vortex we used  $n = 20$  neurons per branch, in all other cases  $n = 10$ . The total number of epochs, including the ones that use BFGS is reported in Tab. 3. Moreover, the loss weights needed to be adjusted as described in Tab. 3 in order to obtain satisfactory accuracies.

## 4 Discussion and Conclusions

In this paper, we presented a new implementation of the PINN concept to solve ODEs/PDEs using NNs - named **dNNsolve**. The main novelties of our implementation reside in two points: *i*) the usage of two types of activation functions - sines and sigmoids - as a more efficient basis to expand the solution and *ii*) the use of a new architecture, tailored in a way to provide the desired expansion in terms of this enlarged basis.

It is well known that a Fourier series can be used to expand any continuous function defined on a compact support, which is the case of the ODEs and PDEs considered in this paper. However, as a generic solution will not be just a periodic function, we expect that a more efficient representation than a bare Fourier expansion can be found by including sigmoid activation functions in the expansion basis. We naively expect that the sigmoids might help the network to capture any secular behaviour featured by the solution, while the sines would take care of any periodic behaviour. We also took

3D results								
PDE	Time(s)	Epochs	$(\alpha_0, \alpha_{\partial\Omega})$	$\log_{10}$				
				$\mathcal{L}$	$\mathcal{L}_{\Omega}$	$\mathcal{L}_0$	$\mathcal{L}_{\partial\Omega}$	$r$
[1]	22.6	706	(10,1)	-5.0	-5.5	-6.3	-5.6	-5.8
[2]	27.5	524	(10,1)	-4.4	-4.6	-5.9	-5.7	-5.8
[3]	15.5	715	(1,1)	-3.7	-3.9	-4.4	-4.4	-4.5
[4]	24.1	750	(10,10)	-4.0	-4.7	-4.5	-4.3	-4.5
[5]	29.0	1484	(10,10)	-2.9	-2.9	-5.0	-4.7	-4.8
[6]	30.0	1546	(1,1)	-2.9	-3.2	-3.8	-3.4	-3.7
[7]	40.2	3277	(1,1)	-1.27	-1.5	-3.7	-2.8	-2.8
[8]	26.1	1640	(1,1)	-1.5	-1.8	-4.0	-2.8	-3.0
[9]	48.9	1165	(10,1)	-2.9	-3.0	-5.3	-4.2	-4.2
[10]	271.0	7389	(1,1)	-1.5	-1.6	-3.5	-2.4	-2.4
[11]	335.5	11232	(1,1)	-1.8	-1.9	-3.6	-2.5	(*)

Table 3: PDEs results in 3D. We do not have a solution to compare with for the case marked with an asterisk, the vorticity equation in Eq. [11].

into account the possible non-linear interplay between a periodic and a non-periodic expansion of the solution using element-wise multiplication of the sine and sigmoid branch. In order to achieve the expansion in this enlarged basis, we introduced a specific architecture, which is pictorially shown in Figure 1. We call this new implementation **dNNsolve**: *d*ual *NN*solve(r) of ODEs and PDEs.

We empirically show that the our expectation is met and the network is able to learn an efficient representation of the solution that most of the times makes use of both the sigmoids and the sines. We also show that in those cases that obviously require only one kind of basis functions (i.e. only sines, sigmoids or multiplication of them), the network is able to adjust the weights accordingly, basically cutting out the redundant branches. On the other hand, given that unused neurons are never exactly set to zero activation, these can bring noise to the solution, compromising the final accuracy. We saw this happening in the oscillon ODE where we had to reduce the number of neurons per branch. This remains an open problem of our architecture, the solution of which we leave for future works.

**dNNsolve** is able to solve a broad range of ODEs in 1D, see App. B, including both first and second order initial value problems. In particular, it successfully solves the damped harmonic oscillator, whose solution rapidly goes to vanishing amplitude, as well as stiff and delayed equations. The accuracies of these solutions, listed in Tab. 1 were computed by comparing with the analytical solutions – where these are available – or with the solutions obtained through a standard solver such as the `odeint` and `ddeint` packages in Python, or `NDSolve` in `Mathematica`. All the root mean squared accuracies are below  $10^{-3}$ , reaching in 1D values  $< 10^{-5}$  for the harmonic oscillator, its damped version and the oscillon profile equation. Interestingly, we were able to solve the oscillon profile equation, see Eq. [10], that is a boundary value problem in 1D, typically solved using the shooting method. We note that tunnelling profiles in phase transitions have been solved with a simple one layer PINN before in [63]. Even though bubble profiles are expected to be particularly well expanded in a sigmoid base, we find that our NN oscillon solution makes use of the Fourier basis, and becomes much more accurate than the typical root mean squared accuracies  $\sim 10^{-3}$  reported in [63]. We plan to expand our study of these kinds of problems in a separate publication.

In 2D, **dNNsolve** was used to solve a variety of PDEs, see App. C, including parabolic, hyperbolic and elliptic types on a rectangular domain  $[0, 1]^2$ . All the accuracies are below  $10^{-4}$  (except for the

heat equation, Eq. [5]), reaching values as small as  $10^{-6}$ , see Tab. 2. Similar results are obtained in the 3D case, see Tab. 3, in which case all the equations are solved on a cube of the form  $[0, 1]^3$ .

As shown in App. A, the use of sine activation functions may introduce local minima in the loss function, making it hard for the NN to converge to the true solution of the ODE/PDE. We addressed this problem by introducing two additional hyperparameters  $\alpha_0$  and  $\alpha_{\partial\Omega}$  that assign different weights to the various parts of the loss function (for initial/boundary value problems), see Eq. (7), in the 2D and 3D cases. In particular, in the 2D case we had to weigh the initial conditions part of the loss,  $\mathcal{L}_0$ , more than the other two pieces in order for `dNNsolve` to converge to the true solution for all the PDEs considered. In the 3D case it was not possible to find a common choice of the hyperparameters  $\alpha$ , hence the optimal values are reported in Tab. 3. Overall, we did not fine-tune the hyperparameters  $\alpha$  in any of the examples presented: even in the 3D case we just picked one combination of  $\alpha_0$  and  $\alpha_{\partial\Omega}$  (choosing their value to be either 1 or 10) that led to an acceptable accuracy. We consider this to be one of the main achievements of our work: `dNNsolve` is able to solve a broad class of differential equations without the need to tune (or with a mild tuning in the 3D case) any hyperparameter. We leave to future work the implementation of an automatic procedure for the optimal choice of the hyperparameters  $\alpha$ . Also, note that the results we have reported are not the best results achievable with `dNNsolve`, as they are obtained using a random set of initialized weights as opposed to the best solution selected out of multiple runs [34].

Finally, one interesting question to ask from the physicist’s point of view is whether these kinds of solvers might be able to replace standard lattice codes for instance to perform cosmological simulations. It has already been shown [47] that NN solvers are able to catch up with finite difference methods both in terms of memory complexity (in 3D) and in terms of time complexity (in 5D). Our method is a further step in the direction of making NN-based PDE solvers competitive with standard methods, since it improves the efficiency of the representation of the PDE solution. In order to become fully competitive though, further improvements need to be implemented, among which: *i*) we need to find a way to reduce the noise coming from superfluous neurons, see discussion above; *ii*) we need a better way for the network to avoid local minima, for instance implementing an automatic tuning of the hyperparameters  $\alpha$ ; *iii*) we need an efficient way to parallelize the computation over multiple CPUs and/or GPUs. We plan to come back to these points in future works.

## Acknowledgments

AW and YW are supported by the ERC Consolidator Grant STRINGFLATION under the HORIZON 2020 grant agreement no. 647995. F.M. thanks Ben Mares for inspiring discussions about topics related to this paper. F.M. is funded by a UKRI/EPSRC Stephen Hawking fellowship, grant reference EP/T017279/1. This work has been partially supported by STFC consolidated grant ST/P000681/1.

## References

- [1] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations*, 2017.
- [2] M. Raissi, P. Perdikaris, and G. E. Karniadakis, *Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations*, 2017.
- [3] G. Felder and I. Tkachev, *Latticeeasy: A program for lattice simulations of scalar fields in an expanding universe*, *Computer Physics Communications* **178** (Jun, 2008) 929–932.
- [4] A. V. Frolov, *Defrost: a new code for simulating preheating after inflation*, *Journal of Cosmology and Astroparticle Physics* **2008** (Nov, 2008) 009.
- [5] J. Sainio, *CUDA-EASY - a GPU Accelerated Cosmological Lattice Program*, *Comput. Phys. Commun.* **181** (2010) 906–912, [[arXiv:0911.5692](#)].
- [6] Z. Huang, *The Art of Lattice and Gravity Waves from Preheating*, *Phys. Rev. D* **83** (2011) 123509, [[arXiv:1102.0227](#)].
- [7] J. Sainio, *PyCOOL - a Cosmological Object-Oriented Lattice code written in Python*, *JCAP* **04** (2012) 038, [[arXiv:1201.5029](#)].

- [8] H. L. Child, J. T. Giblin, Jr, R. H. Ribeiro, and D. Seery, *Preheating with Non-Minimal Kinetic Terms*, *Phys. Rev. Lett.* **111** (2013) 051301, [[arXiv:1305.0561](https://arxiv.org/abs/1305.0561)].
- [9] K. Clough, P. Figueras, H. Finkel, M. Kunesch, E. A. Lim, and S. Tunyasuvunakool, *GRChombo : Numerical Relativity with Adaptive Mesh Refinement*, *Class. Quant. Grav.* **32** (2015) 24, [[arXiv:1503.03436](https://arxiv.org/abs/1503.03436)].
- [10] V. Mewes, Y. Zlochower, M. Campanelli, I. Ruchlin, Z. B. Etienne, and T. W. Baumgarte, *Numerical relativity in spherical coordinates with the einstein toolkit*, *Physical Review D* **97** (Apr, 2018).
- [11] J. T. Giblin and A. J. Tishue, *Preheating in Full General Relativity*, *Phys. Rev. D* **100** (2019), no. 6 063543, [[arXiv:1907.10601](https://arxiv.org/abs/1907.10601)].
- [12] M. Vogelsberger, F. Marinacci, P. Torrey, and E. Puchwein, *Cosmological Simulations of Galaxy Formation*, *Nature Rev. Phys.* **2** (2020), no. 1 42–66, [[arXiv:1909.07976](https://arxiv.org/abs/1909.07976)].
- [13] W. R. Inc., “Mathematica, Version 12.2.”
- [14] MATLAB, *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.
- [15] C. Multiphysics, *Introduction to comsol multiphysics®*, COMSOL Multiphysics, Burlington, MA, accessed Feb **9** (1998) 2018.
- [16] Maplesoft, a division of Waterloo Maple Inc., “Maple.”
- [17] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, and O. Sander, *The dune framework: Basic concepts and recent developments*, 2020.
- [18] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, *The fenics project version 1.5*, *Archive of Numerical Software* **3** (2015), no. 100.
- [19] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. C. V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, *MFEM: A modular finite element library*, *Computers & Mathematics with Applications* (2020).
- [20] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, *Mathematics of control, signals and systems* **2** (1989), no. 4 303–314.
- [21] K. Hornik, *Approximation capabilities of multilayer feedforward networks*, *Neural Networks* **4** (1991), no. 2 251–257.
- [22] A. Pinkus, *Approximation theory of the mlp model in neural networks*, *Acta Numerica* **8** (1999) 143–195.
- [23] H. Lee and I. Kang, *Neural algorithm for solving differential equations*, *Journal of Computational Physics* **91** (1990) 110–131.
- [24] I. Lagaris, A. Likas, and D. Fotiadis, *Artificial neural networks for solving ordinary and partial differential equations*, *IEEE Transactions on Neural Networks* **9** (1998), no. 5 987–1000.
- [25] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, *Neural-network methods for boundary value problems with irregular boundaries*, *IEEE Transactions on Neural Networks* **11** (2000), no. 5 1041–1049.
- [26] I. E. Lagaris, A. Likas, and D. I. Fotiadis, *Artificial neural network methods in quantum mechanics*, *Comput. Phys. Commun.* **104** (1997) 1–14, [[quant-ph/9705029](https://arxiv.org/abs/quant-ph/9705029)].
- [27] K. Rudd, G. Di Muro, and S. Ferrari, *A constrained backpropagation approach for the adaptive solution of partial differential equations*, *IEEE transactions on neural networks and learning systems* **25** (2013), no. 3 571–584.

- [28] K. Rudd and S. Ferrari, *A constrained integration (cint) approach to solving partial differential equations using artificial neural networks*, *Neurocomputing* **155** (2015) 277–285.
- [29] S. Ferrari and M. Jensenius, *A constrained optimization approach to preserving prior knowledge during incremental training*, *IEEE Transactions on Neural Networks* **19** (2008), no. 6 996–1009.
- [30] B. Galerkin, *On electrical circuits for the approximate solution of the laplace equation*, *Vestnik Inzh* **19** (1915) 897–908.
- [31] S. G. Michlin and E. Heyn, *Variationsmethoden der mathematischen Physik*. Akademie-Verlag, 1962.
- [32] M. M. Vajnberg, *Variational method and method of monotone operators in the theory of nonlinear equations*. Wiley, 1973.
- [33] J. Sirignano and K. Spiliopoulos, *Dgm: A deep learning algorithm for solving partial differential equations*, *Journal of Computational Physics* **375** (Dec, 2018) 1339–1364.
- [34] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, *Deepxde: A deep learning library for solving differential equations*, 2020.
- [35] A. Koryagin, R. Khudorozkov, and S. Tsimfer, *Pydens: a python framework for solving differential equations with neural networks*, 2019.
- [36] A. D. Jagtap, K. Kawaguchi, and G. E. Karniadakis, *Adaptive activation functions accelerate convergence in deep and physics-informed neural networks*, *Journal of Computational Physics* **404** (2020) 109136.
- [37] A. D. Jagtap, K. Kawaguchi, and G. E. Karniadakis, *Locally adaptive activation functions with slope recovery term for deep and physics-informed neural networks*, 2020.
- [38] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, *Accelerating eulerian fluid simulation with convolutional networks*, 2017.
- [39] N. Geneva and N. Zabaras, *Modeling the dynamics of pde systems with physics-constrained deep auto-regressive networks*, *Journal of Computational Physics* **403** (Feb, 2020) 109056.
- [40] N. Thuerey, K. Weißenow, L. Prantl, and X. Hu, *Deep learning methods for reynolds-averaged navier–stokes simulations of airfoil flows*, *AIAA Journal* **58** (Jan, 2020) 25–36.
- [41] Y. Yang and P. Perdikaris, *Adversarial uncertainty quantification in physics-informed neural networks*, *Journal of Computational Physics* **394** (Oct, 2019) 136–152.
- [42] E. Samaniego, C. Anitescu, S. Goswami, V. Nguyen-Thanh, H. Guo, K. Hamdia, X. Zhuang, and T. Rabczuk, *An energy approach to the solution of partial differential equations in computational mechanics via machine learning: Concepts, implementation and applications*, *Computer Methods in Applied Mechanics and Engineering* **362** (Apr, 2020) 112790.
- [43] W. E, J. Han, and A. Jentzen, *Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations*, *Communications in Mathematics and Statistics* **5** (Nov, 2017) 349–380.
- [44] P. Chaudhari, A. Oberman, S. Osher, S. Soatto, and G. Carlier, *Deep relaxation: partial differential equations for optimizing deep neural networks*, 2017.
- [45] C. Beck, W. E, and A. Jentzen, *Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations*, *Journal of Nonlinear Science* **29** (Jan, 2019) 1563–1619.
- [46] M. Fujii, A. Takahashi, and M. Takahashi, *Asymptotic expansion as prior knowledge in deep learning method for high dimensional bsdes*, 2019.

- [47] V. I. Avrutskiy, *Neural networks catching up with finite differences in solving partial differential equations in higher dimensions*, *Neural Computing and Applications* **32** (2020), no. 17 13425–13440.
- [48] T. Dockhorn, *A discussion on solving partial differential equations using neural networks*, 2019.
- [49] J. Bedford and M. Bevis, *Greedy automatic signal decomposition and its application to daily gps time series*, *Journal of Geophysical Research: Solid Earth* **123** (2018), no. 8 6992–7003, [<https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2017JB014765>].
- [50] A. Silvescu, *Fourier neural networks, IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)* **1** (1999) 488–491 vol.1.
- [51] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, *Implicit neural representations with periodic activation functions*, 2020.
- [52] J. Zhang, G. G. Walter, Y. Miao, and W. N. W. Lee, *Wavelet neural networks for function learning*, *IEEE transactions on Signal Processing* **43** (1995), no. 6 1485–1497.
- [53] J. M. Sopena, E. Romero, and R. Alquezar, *Neural networks with periodic and monotonic activation functions: a comparative study in classification problems*, in *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 1, pp. 323–328 vol.1, 1999.
- [54] G. Parascandolo, H. Huttunen, and T. Virtanen, *Taming the waves: sine as activation function in deep neural networks*, .
- [55] Y. Shin, *On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes*, *Communications in Computational Physics* **28** (Jun, 2020) 2042–2074.
- [56] R. van der Meer, C. Oosterlee, and A. Borovykh, *Optimally weighted loss functions for solving pdes with neural networks*, 2020.
- [57] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014).
- [58] J. E. Dennis and J. J. Moré, *A characterization of superlinear convergence and its application to quasi-Newton methods*, *Math. Comp.* **28** (1974) 549–560.
- [59] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. <http://www.deeplearningbook.org>.
- [60] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., *Tensorflow: A system for large-scale machine learning*, in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.
- [61] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, *Tensorflow distributions*, 2017.
- [62] Z.-Q. J. Xu, Y. Zhang, and Y. Xiao, *Training behavior of deep neural network in frequency domain*, 2019.
- [63] M. L. Piscopo, M. Spannowsky, and P. Waite, *Solving differential equations with neural networks: Applications to the calculation of cosmological phase transitions*, *Phys. Rev. D* **100** (2019), no. 1 016002, [[arXiv:1902.05563](https://arxiv.org/abs/1902.05563)].

## A Comments on $\alpha$ hyperparameters

In this section we provide some empirical evidence about the advantages of using  $\alpha$  hyperparameters to give different weights to the various loss function contributions. To do this, we discuss a couple of examples, focusing on the periodic branch of our architecture. We will consider the simplest Fourier NN with 1 hidden layer and 1 neuron per input variable.

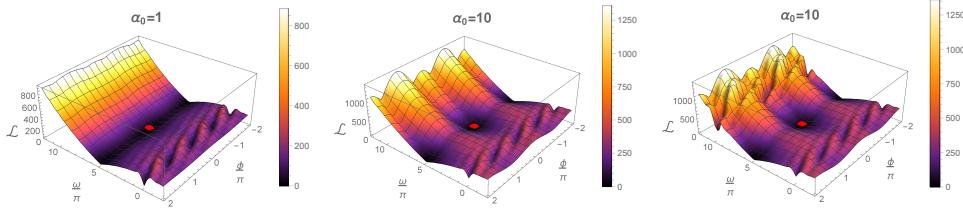


Figure 4: 2D loss function  $\mathcal{L}(\omega, \phi)$  obtained from sampling bulk points on a grid of step  $\Delta t = 0.05$  (left, center) and  $\Delta t = 0.1$  (right). Left and central plots show the shape of the loss surface when  $\alpha_0 = 1$  and  $\alpha_0 = 10$  respectively. These plots do not reveal any appreciable difference with the analytic result coming from the continuous limit. The right plot shows how artificial local minima can arise from poor sampling.

**1D harmonic oscillator** We want to get some intuition about the loss space when using a sine activation function. In order to do that, we consider a NN having a single sine neuron with a 1D input:

$$\hat{u} = d \sin(\omega t + \phi) \quad (12)$$

We try to understand what is the shape of the loss function if we look for solutions of the following harmonic oscillator ODE:

$$\begin{cases} u'' + (5\pi)^2 u = 0 \\ u(0) = 0 \\ u'(0) = 10\pi \end{cases} \quad t \in [0, 1]$$

. According to our prescriptions, the loss function in the bulk is given by

$$\begin{aligned} \mathcal{L}_\Omega^2 &= \frac{1}{n_\Omega} \sum_i [\hat{u}'' + (5\pi)^2 \hat{u}]^2 \\ &\sim \frac{d^2}{(\Delta x)} [(5\pi)^2 - \omega^2]^2 \int dt \sin^2(\omega t + \phi) \\ &= \frac{d^2}{(\Delta x)} [(5\pi)^2 - \omega^2]^2 \left( \frac{2\omega + \sin(2\phi) - \sin[2(\omega + \phi)]}{4\omega} \right), \end{aligned} \quad (13)$$

while the loss for the initial condition is

$$\begin{aligned} \mathcal{L}_0^2 &= \frac{1}{2n_0} \sum_j \left( [\hat{u}(0)]^2 + [\hat{u}'(0)]^2 - 5\pi^2 \right) \\ &= \frac{1}{2} \left( d^2 \sin^2(\phi) + [d\omega \cos(\phi) - 10\pi]^2 \right). \end{aligned} \quad (14)$$

The total loss is given by the weighted sum:

$$\mathcal{L} = \mathcal{L}_\Omega + \alpha_0 \mathcal{L}_0. \quad (15)$$

In order to see the relation between the different pieces of the loss function and understand the effects of discretization, we provide some plots of  $\mathcal{L}$  related to this simple example. Let us first fix the amplitude parameter to its minimum  $d = 2$  and plot the 2D loss surface in Figure 4. These plots are obtained using the discrete definition of the loss, for different values of  $\alpha_0 = 1, 10$  and different numbers of bulk points. Without loss of generality, we sample the bulk points on an equally spaced grid. We can clearly see the presence of local minima near  $\omega = 0$  coming from the bulk part of the loss. From the central plot we can infer that setting  $\alpha_0 \gg 1$  sharpens the location of the global minima located in  $\omega = 5\pi, \phi = 0 + 2\pi k, k \in \mathbb{Z}$ . Moreover, comparing the results obtained using  $\Delta t = 0.05$  and  $\Delta t = 0.1$ , we see that choosing a too wide gridstep may lead to the formation of spurious local minima. Despite its simplicity, this example is able to highlight the problems related to NNs with periodic activation functions. On top of local minima coming from the discretization procedure, the loss is intrinsically fulfilled with functions having a huge number of local minima, e.g. cardinal sine.

Let us now also investigate the rôle of the amplitude parameter  $d$ . The top panels of Figure 5 show the bulk and IC part of the loss function while the total losses, considering  $\alpha_0 = 1, 10$ , are shown

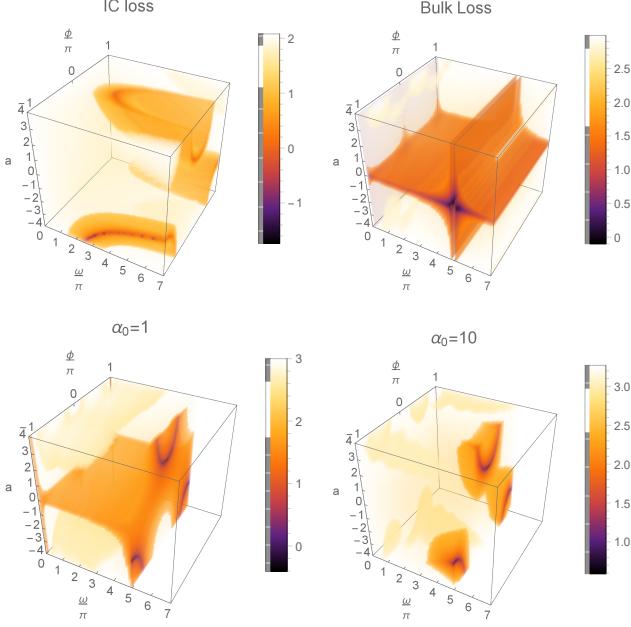


Figure 5: 3D loss volume,  $\mathcal{L}(\omega, \phi, d)$ , obtained from sampling bulk points on a grid of step  $\Delta t = 0.1$ . Top plots show the IC (left) and bulk contributions (right). Bottom plot show the total loss function computed considering  $\alpha_0 = 1$  (left) and  $\alpha_0 = 10$  (right). To simplify viewing we clip the surface  $\omega < 5\pi$ . These plots do not reveal any appreciable difference with the analytic result coming from the continuous limit.

in the bottom panels. As before, these plots are obtained using a grid with  $\Delta t = 0.1$ . From the top plots we can see that IC and bulk loss do not share their local minima. Compared to the previous analysis, the bulk loss exhibits a new local minimum at  $d = 0$ . This comes from the fact that the wave equation is homogeneous and allows for vanishing solutions. Given that  $\mathcal{L}_\Omega$  linearly depends on  $d$ , it is very easy for the network to minimize the  $d$  direction. This, coupled to the fact that on average  $\mathcal{L}_\Omega > \mathcal{L}_0$ , implies that in the early stages of training the network parameters will rapidly reach the  $d = 0$  surface. If we do not introduce  $\alpha$  hyperparameters, this behaviour becomes a serious problem of the training process. Indeed, the bottom-left plot of Figure 5 shows that, choosing  $\alpha_0 = 1$ ,  $d = 0$  is a flat local minimum of the loss function, i.e.  $\nabla_{\omega, \phi} \mathcal{L}|_{d=0} = (0, 0)$ . On the other hand, according to what we have seen before, setting  $\alpha_0 \gg 1$  sharpens the location of the global minima (bottom-right plot). This result should not be surprising at all: an ODE defined on a certain domain admits infinite solutions and the result of an ODE problem becomes unique if and only if we provide the initial conditions. Therefore, we must set  $\alpha_0 \gg 1$  to counteract the loss hierarchy  $\mathcal{L}_\Omega > \mathcal{L}_0$ . Indeed, also from a theoretical point of view, the minimization of the bulk loss becomes really informative only if it takes place after or together with the minimization of  $\mathcal{L}_0$ .

**2+1D wave equation** We focus on a slightly more complex toy model given by solving the 2+1D wave equation (see Eq. [1]) with three sine neurons. A simplified version of our sinusoidal branch would be given by:

$$\hat{u} = \sin(\omega_t t + \phi_t) \sin(\omega_x x + \phi_x) \sin(\omega_y y + \phi_y). \quad (16)$$

We use this network to solve the following PDE:

$$\begin{cases} \partial_{tt}^2 u - (\partial_{xx}^2 u + \partial_{yy}^2 u) = 0 \\ u(0, x, y) = \sin(3\pi x) \sin(4\pi y) \\ \partial_t u(0, x, y) = 0 \\ u|_{\partial\Omega} = 0 \end{cases} \quad \Omega = [0, 1]^3.$$

According to the definition given in Eq. (7), the parts of the loss function associated to the bulk, the initial condition and the boundary are given by:

$$\begin{aligned}
\mathcal{L}_\Omega^2 &= \frac{1}{n_\Omega} \sum_i [\partial_{tt}^2 \hat{u} - (\partial_{xx}^2 \hat{u} + \partial_{yy}^2 \hat{u})]^2 \\
&\sim \frac{1}{V(\Omega)} \int_\Omega dx dy dt [\partial_{tt}^2 \hat{u} - (\partial_{xx}^2 \hat{u} + \partial_{yy}^2 \hat{u})]^2 \\
&= \frac{(\omega_x^2 + \omega_y^2 - \omega_t^2)^2}{V(\Omega)} \int_\Omega dx dy dt [\sin(\omega_t t + \phi_t) \sin(\omega_x x + \phi_x) \sin(\omega_y y + \phi_y)]^2 \\
&= \frac{(\omega_x^2 + \omega_y^2 - \omega_t^2)^2}{V(\Omega)} \prod_{i=t,x,y} \left( \frac{2\omega_i + \sin(2\phi_i) - \sin[2(\omega_i + \phi_i)]}{4\omega_i} \right), 
\end{aligned} \tag{17}$$

$$\begin{aligned}
\mathcal{L}_0^2 &= \frac{1}{2n_0} \sum_j ([\hat{u}(t_0) - \sin(3\pi x) \sin(4\pi y)]^2 + [[\partial_t \hat{u}](t_0)])^2 \\
&\sim \frac{1}{2A(\Omega_0)} \int_{\Omega_0} dx dy ([\hat{u}(t_0) - \sin(3\pi x) \sin(4\pi y)]^2 + [[\partial_t \hat{u}](t_0)])^2 \\
&= \frac{1}{2A(\Omega_0)} \left[ \int_{\Omega_0} dx dy [\sin(\phi_t) \sin(\omega_x x + \phi_x) \sin(\omega_y y + \phi_y) - \sin(3\pi x) \sin(4\pi y)]^2 + \right. \\
&\quad \left. + \int_{\Omega_0} dx dy [\omega_t \cos(\phi_t) \sin(\omega_x x + \phi_x) \sin(\omega_y y + \phi_y)]^2 \right] \\
&= \frac{1}{2A(\Omega_0)} \left[ \frac{1}{4} - 24\pi^2 \sin(\phi_t) \left( \frac{\sin(\phi_x) + \sin(\phi_x + \omega_x)}{(9\pi^2 - \omega_x^2)} \right) \left( \frac{\sin(\phi_y) + \sin(\phi_y + \omega_y)}{(16\pi^2 - \omega_6^2)} \right) \right. \\
&\quad \left. + (\sin^2(\phi_t) + \omega_t^2 \cos^2(\phi_t)) \prod_{i=x,y} \left( \frac{2\omega_i + \sin(2\phi_i) - \sin[2(\omega_i + \phi_i)]}{4\omega_i} \right) \right], 
\end{aligned} \tag{18}$$

$$\begin{aligned}
\mathcal{L}_{\partial\Omega}^2 &= \frac{1}{n_{\partial\Omega}} \sum_k [\hat{u}|_{\partial\Omega}]^2 \\
&\sim \frac{1}{4A(\partial\Omega)} \left( \int_{x=0} dy dt [\hat{u}]^2 + \int_{x=1} dy dt [\hat{u}]^2 + \int_{y=0} dy dt [\hat{u}]^2 + \int_{y=1} dy dt [\hat{u}]^2 \right) \\
&= \frac{1}{4A(\partial\Omega)} \left( (\sin^2(\phi_x) + \sin^2(\omega_x + \phi_x)) \int dy dt (\sin^2(\omega_t t + \phi_t) \sin^2(\omega_y y + \phi_y)) \right. \\
&\quad \left. + (\sin^2(\phi_y) + \sin^2(\omega_y + \phi_y)) \int dy dt (\sin^2(\omega_t t + \phi_t) \sin^2(\omega_x x + \phi_x)) \right) \\
&= \frac{1}{4A(\partial\Omega)} \left\{ \left( \frac{2\omega_t + \sin(2\phi_t) - \sin[2(\omega_t + \phi_t)]}{4\omega_t} \right) \times \right. \\
&\quad \left. \left[ (\sin^2(\phi_x) + \sin^2(\omega_x + \phi_x)) \left( \frac{2\omega_y + \sin(2\phi_y) - \sin[2(\omega_y + \phi_y)]}{4\omega_y} \right) \right. \right. \\
&\quad \left. \left. + (\sin^2(\phi_y) + \sin^2(\omega_y + \phi_y)) \left( \frac{2\omega_x + \sin(2\phi_x) - \sin[2(\omega_x + \phi_x)]}{4\omega_x} \right) \right] \right\}. 
\end{aligned} \tag{19}$$

We plot  $\mathcal{L}_\Omega$ ,  $\mathcal{L}_0$  and  $\mathcal{L}_{\partial\Omega}$  in Figure 6, fixing the phase parameters at their minima. These results show that there is a natural hierarchy, i.e.  $\mathcal{L}_\Omega \gg \mathcal{L}_0, \mathcal{L}_{\partial\Omega}$ , that can be explained by the effect of derivation on Fourier NN: each derivative provides a  $\omega$  power in the loss function. This results in a polynomial behaviour of  $\mathcal{L}_\Omega$  in the frequency space, while initial conditions and boundary conditions may be at most linear. Given that the initial conditions of the parameters are randomly chosen, this implies that, on average,  $\mathcal{L}_\Omega$  will be the major contribution to the loss function in the early stages of training.

Going back to our example, this means that training will lead the parameters toward the surface  $(\omega_x^2 + \omega_y^2 - \omega_t^2) = 0$  that is clearly visible in the left plot of Figure 1. From then on, the trajectory in

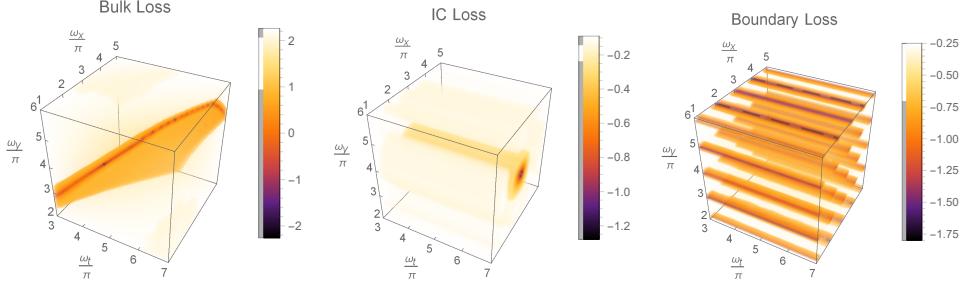


Figure 6: Continuous limit of the different contributions to the loss function setting phase weights to their minima:  $\mathcal{L}_\Omega$  (left),  $\mathcal{L}_0$  (center) and  $\mathcal{L}_{\partial\Omega}$  (right). Discrete versions computed on  $11^3$  grid points do not show any significant difference inside this volume. The legend bars refer to  $\log_{10}(\mathcal{L})$ .

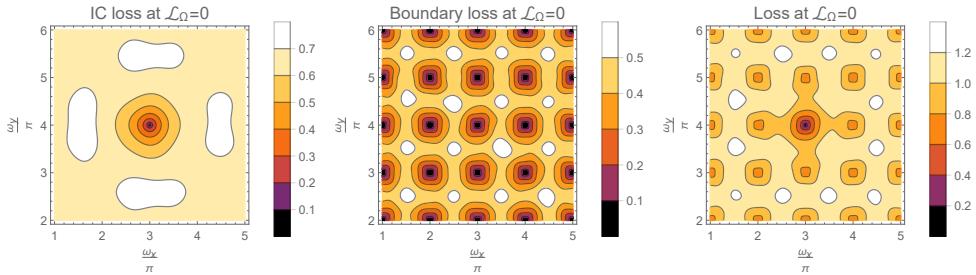


Figure 7: These 1970s typical tiles show the 2D loss function that appears after  $\mathcal{L}_\Omega$  minimization given by  $(\omega_x^2 + \omega_y^2 - \omega_t^2) = 0$ . We consider a grid of  $11^2$  points. While the contribution coming from IC (left) shows a single minimum, the boundary contribution is fulfilled with equally spaced minima (center). The total residual loss (right) is then characterised by the presence of infinitely many local minima surrounding the true solution. This makes the NN training extremely hard and the system gets usually stuck in a local minimum.

the parameter space will not be able to leave the surface and there is a high probability of getting stuck in one of the minima of  $\mathcal{L}_{\partial\Omega}$ , see Figure 7. One way to overcome this problem is to impose some hierarchy among the different contributions to the loss function. This can be done by setting the pieces that carry more information about the solution to higher values. In the specific case analyzed in this section, this results in setting  $\alpha_0 \gg 1$ .

## B 1D ODEs under study

[1] Mathieu equation:

$$\begin{cases} u''(t) + (a - 2q \cos(2t))u(t) = 0 & a = 1, q = 0.2 \\ u(0) = 1 \\ u'(0) = 0 \end{cases}$$

[2] Decaying exponential:

$$\begin{cases} u'(t) + \beta u(t) = 0 & \beta = 0.52 \\ u(0) = 1 \end{cases}$$

with analytical solution

$$u(t) = e^{-\beta t}$$

[3] Harmonic oscillator:

$$\begin{cases} u''(t) + \omega^2 u(t) = 0 & \omega = 5 \\ u(0) = 1 \\ u'(0) = 0 \end{cases}$$

with analytical solution

$$u(t) = \cos(\omega t)$$

[4] Damped harmonic oscillator:

$$\begin{cases} u''(t) + \beta u'(t) + \omega^2 u(t) = 0 \\ u(0) = 1 \\ u'(0) = 0 \end{cases}$$

with analytical solution

$$u(t) = e^{-\beta t/2} \left( \cos(f t) + \frac{\beta}{2f} \sin(f t) \right), \quad f \equiv \sqrt{\omega^2 - \beta^2/4}$$

[5] Linear function:

$$\begin{cases} u'(t) - 1 = 0 \\ u(0) = 1 \end{cases}$$

with analytical solution

$$u(t) = 1 + t$$

[6] Delay equation:

$$\begin{cases} u'(t) - \beta u(t) + u(t-d) = 0, u(t|t < 0) = t-1, \\ u(0) = 1 \end{cases} \quad d = 1$$

[7] Stiff equation::

$$\begin{cases} u'(t) + 21 u(t) - e^{-t} = 0 \\ u(0) = 1 \end{cases}$$

with analytical solution

$$u(t) = \frac{1}{20} (e^{-t} + 19e^{-21t})$$

[8] Gaussian:

$$\begin{cases} u'(t) + 2 b t u(t) = 0 \\ u(0) = 1 \end{cases} \quad b = 0.1$$

with analytical solution

$$u(t) = e^{-bt^2}$$

[9] Two frequencies:

$$\begin{cases} u''(t) + u(t) + A_1 \cos(\omega_1 t) + A_2 \sin(\omega_2 t), \\ u(0) = 1 \\ u'(0) = 0 \end{cases} \quad A_1 = 2, A_2 = 6, \omega_1 = 5, \omega_2 = 10$$

with analytical solution

$$u(t) = \frac{1}{132} (121 \cos(t) + 11 \cos(5t) - 80 \sin(t) + 8 \sin(10t))$$

[10] Oscillon Profile equation:

$$\begin{cases} u''(t) + \frac{d-1}{r} u'(t) + m^2 u(t) - 2 u^3(t) = 0, \\ u' + mu = 0 \quad \text{when } t \rightarrow \infty \\ u'(0) = 0 \end{cases} \quad d = 1$$

with analytical solution

$$u(t) = \frac{m}{\cosh(m t)}$$

## C 2D PDEs under study

[1] Wave equation:

$$\begin{cases} \partial_{tt}^2 u - \partial_{xx}^2 u = 0 \\ u(0, x) = \sin(3\pi x) \\ \partial_t u(0, x) = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \cos(3\pi t) \sin(3\pi x)$$

[2] Wave equation:

$$\begin{cases} \partial_{tt}^2 u - \partial_{xx}^2 u = 0 \\ u(0, x) = \sin(3\pi x) \\ \partial_t u(0, x) = 0 \\ \partial_x u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \cos(3\pi t) \cos(3\pi x)$$

[3] Traveling Wave equation:

$$\begin{cases} \partial_t u - \partial_x u = 0 \\ u(0, x) = \sin(2\pi x) \\ u|_{\partial\Omega} = \sin(2\pi t) \end{cases}$$

with analytical solution

$$u(t, x) = \cos(2\pi(t + x))$$

[4] Heat equation 1:

$$\begin{cases} \partial_t u - 0.05 \partial_{xx}^2 u = 0 \\ u(0, x) = \sin(3\pi x) \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \sin(3\pi x) e^{-0.05(3\pi)^2 t}$$

[5] Heat equation 2:

$$\begin{cases} \partial_t u - 0.01 \partial_{xx}^2 u = 0 \\ u(0, x) = 2 \sin(9\pi x) + 0.3 \sin(4\pi x) \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = 2 \sin(9\pi x) e^{-0.01(9\pi)^2 t} - 0.3 \sin(4\pi x) e^{-0.01(4\pi)^2 t}$$

[6] Heat equation 3:

$$\begin{cases} \partial_t u - 0.05 \partial_{xx}^2 u = 0 \\ u(0, x) = \sin(3\pi x) \\ \partial_x u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \cos(3\pi x) e^{-0.05(3\pi)^2 t}$$

[7] Poisson equation 1:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u + 2\pi^2 \sin(\pi t) \sin(\pi x) = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \sin(\pi x) \sin(\pi t)$$

[8] Poisson equation 2:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u + 10(t-1) \cos(5x) + 25(t-1)(x-1) \sin(5x) = 0 \\ u(0, x) = (1-x) \sin(5x) \\ u(1, x) = u(t, 0) = u(t, 1) = 0 \end{cases}$$

with analytical solution

$$u(t, x) = (1-t)(1-x) \sin(5x)$$

[9] Advection diffusion equation:

$$\begin{cases} \partial_t u - \frac{1}{4} \partial_{xx}^2 u = 0 \\ u(0, x) = \frac{1}{4} \sin(\pi x) \\ u(t, 0) = u(t, 1) = 0 \end{cases}$$

with analytical solution

$$u(t, x) = \frac{1}{4} e^{-\frac{1}{4}\pi^2 t} \sin(\pi x)$$

[10] Burger's equation:

$$\begin{cases} \partial_t u + u \partial_x u - \frac{1}{4} \partial_{xx}^2 u = 0 \\ u(0, x) = x(1-x) \\ u(t, 0) = u(t, 1) = 0 \end{cases}$$

[11] Parabolic equation on unit disk:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u - 4 = 0 \\ u|_{\partial\Omega} = 1 \end{cases}$$

with analytical solution

$$u(t, x) = \frac{1}{4} e^{-\frac{1}{4}\pi^2 t} \sin(\pi x)$$

[12] Poisson equation on unit disk:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u - e^{-(t^2+10x^2)} = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

## D 3D PDEs under study

[1] Wave equation:

$$\begin{cases} \partial_{tt}^2 u - (\partial_{xx}^2 u + \partial_{yy}^2 u) = 0 \\ u(0, x, y) = \sin(\pi x) \sin(\pi y) \\ \partial_t u(0, x, y) = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x, y) = \cos(\sqrt{2}\pi t) \sin(\pi x) \sin(\pi y)$$

[2] Wave equation:

$$\begin{cases} \partial_{tt}^2 u - (\partial_{xx}^2 u + \partial_{yy}^2 u) = 0 \\ u(0, x, y) = \sin(3\pi x) \sin(4\pi y) \\ \partial_t u(0, x, y) = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x, y) = \cos(5\pi t) \sin(3\pi x) \sin(4\pi y)$$

[3] Traveling wave equation:

$$\begin{cases} \partial_t u - \frac{1}{5}(\partial_x u + \partial_y u) = 0 \\ u(0, x, y) = \sin(3\pi x + 2\pi y) \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$u(t, x, y) = \sin(3\pi x + 2\pi y + \pi t)$$

[4] Heat equation 1:

$$\begin{cases} \partial_t u - (\partial_{xx}^2 + \partial_{yy}^2)u = 0 \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$u(t, x, y) = e^{x+y+2t}$$

[5] Heat equation 2:

$$\begin{cases} \partial_t u - (\partial_{xx}^2 + \partial_{yy}^2)u = 0 \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$u(t, x, y) = (1-y)e^{x+t}$$

[6] Poisson equation 1:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u + \partial_{yy}^2 u + 3\pi^2 \sin(\pi t) \sin(\pi x) \sin(\pi y) = 0 \\ u|_{\partial\Omega} = 0 \end{cases}$$

with analytical solution

$$u(t, x, y) = \sin(\pi t) \sin(\pi x) \sin(\pi y)$$

[7] Poisson equation 2:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u + \partial_{yy}^2 u - 6 = 0 \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$u(t, x, y) = u(t, x, y) = t^2 + x^2 + y^2$$

[8] Poisson equation 3:

$$\begin{cases} \partial_{tt}^2 u + \partial_{xx}^2 u + \partial_{yy}^2 u - 6 = 0 \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$u(t, x, y) = t^2 + x^2 - y^2$$

[9] Taylor-Green vortex:

$$\begin{cases} \partial_t u + u \partial_x u + v \partial_y u + \frac{1}{2} e^{-4t} \sin 2x - (\partial_{xx} u + \partial_{yy}) u = 0 \\ \partial_t v + u \partial_x v + v \partial_y v + \frac{1}{2} e^{-4t} \sin 2y - (\partial_{xx} v + \partial_{yy} v) = 0 \\ \partial_x u + \partial_y v = 0 \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$\begin{cases} u(t, x, y) = \cos(x) \sin(y) e^{-2t} \\ v(t, x, y) = \sin(x) \cos(y) e^{-2t} \end{cases}$$

[10] Lamb-Oseen vortex:

$$\begin{cases} \omega = \partial_x v - \partial_y u \\ \partial_t \omega + u \partial_x \omega + v \partial_y \omega - 5 \cdot 10^{-3} (\partial_{xx} \omega + \partial_{yy} \omega) \omega = 0 \\ \partial_x u + \partial_y v = 0 \end{cases}$$

where

$$\begin{cases} w(0, x, y) = \frac{1}{4\pi t} \exp \left[ -\frac{x^2 + y^2}{4t} \right] \\ \text{Dirichlet BC} \end{cases}$$

with analytical solution

$$\begin{cases} u(t, x, y) = -\frac{y}{2\pi(x^2 + y^2)} \left( 1 - \exp \left[ -\frac{x^2 + y^2}{4t} \right] \right) \\ v(t, x, y) = \frac{x}{2\pi(x^2 + y^2)} \left( 1 - \exp \left[ -\frac{x^2 + y^2}{4t} \right] \right) \end{cases}$$

[11] Vorticity equation:

$$\begin{cases} \omega = \partial_x v - \partial_y u \\ \partial_t \omega + u \partial_x \omega + v \partial_y \omega - 5 \cdot 10^{-3} (\partial_{xx} \omega + \partial_{yy} \omega) \omega - 0.75 [\sin(2\pi(x+y)) + \cos(2\pi(x+y))] = 0 \\ \partial_x u + \partial_y v = 0 \end{cases}$$

where

$$\begin{cases} w(0, x, y) = \pi [\cos(3\pi x) - \cos(3\pi y)] \\ u(t, 0, y) = u(t, 1, y) \\ u(t, x, 0) = u(t, x, 1) \\ u(t, 0, y) = u(t, 1, y) \\ v(t, x, 0) = u(t, x, 1) \end{cases}$$