POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Bridging Traditional and Machine Learning-based Algorithm for solving PDEs: The Random Feature Method

Advanced Programming for Scientific Computing

Authors: Viola Brenzone and Marco Rapetti

Advisor: Paolo Zunino, Andrea Manzoni

Co-advisors: Nicola Rares Franco

Academic year: 2022-2023

Contents

1. Physics Informed Machine Learning methods to solve PDEs

Algorithms for solving Partial Differential Equations (PDEs) are one of the most studied subjects in scientific computing. Finite difference, finite element, spectral methods, and a host of other methodologies have been proposed and studied with great success. However, more recently, solving partial differential equations (PDEs), e.g., in the standard differential form or in the integral form, via deep learning has emerged as a potentially new subfield under the name of Scientific Machine Learning (SciML). In particular, we can replace traditional numerical discretization methods with a neural network that approximates the solution of a PDE. To obtain an approximate solution of a PDE via deep learning, a key step is to constrain the neural network to minimize the residual of the PDE, and several approaches have been proposed to accomplish this. The goal of this project is to implement the Random Feature Method, proposed in [1]. This new method represents a natural bridge between traditional and machine learning-based algorithms for solving partial differential equations. To better understand the idea behind this new proposed methodology, in the next sessions we briefly describe the main features of a machine-learning based algorithm that would be the basis for our implementation.

1.1. Physiscs Informed Neural Network (PINN)

Compared to traditional mesh-based methods, such as the finite difference method (FDM) and the finite element method (FEM), deep learning could be a mesh-free approach by taking advantage of automatic differentiation and could break the curse of dimensionality. Physics-Informed Neural Networks (PINNs) approximate PDEs in the strong form directly; in this form, automatic differentiation could be used directly to avoid truncation errors and numerical quadrature errors of variational forms. An attractive feature of PINNs is that it can be used to solve inverse problems with a minimum change of the code for forward problems.

1.1.1 Deep Neural Network

Mathematically, a deep neural network is a particular choice of compositional function. The simplest neural network is the Feed Forward Neural Network (FNN), which applies linear and non-linear transformations to the inputs recursively.
Let $\mathcal{N}^L(\mathbf{x})$: $\mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ be a L-layer neural network, or a $(L-1)$-hidden layer neural network, with N_l neurons in the l-th layer ($N_0 = d_{in}$, $N_L = d_{out}$). Let us denote the weight matrix and bias vector in the l-th layer by $\boldsymbol{W}^l \in \mathbb{R}^{N_l \times N_l}$ and $\mathbf{b}^l \in \mathbb{R}^{N_l}$, respectively. Given a nonlinear activation function σ, which is applied element-wise, the FNN is recursively defined as follows:

- input layer: $\mathcal{N}^0(\mathbf{x}) = \mathbf{x} \in \mathbb{R}^{d_{in}}$
- hidden layer: $\mathcal{N}^l(\mathbf{x}) = \sigma(\boldsymbol{W}^l \mathcal{N}^{l-1}(\mathbf{x}) + \mathbf{b}^l) \in \mathbb{R}^{N_l}$
- output layer: $\mathcal{N}^L(\mathbf{x}) = \boldsymbol{W}^L \mathcal{N}^L(\mathbf{x}) + \mathbf{b}^L \in \mathbb{R}^{d_{out}}$

1.1.2 Automatic Differentiation

In PINNs, it is required to compute the derivatives of the network outputs with respect to the network inputs. In deep learning, the derivatives are evaluated using backpropagation, a technique of automatic differentiation. Considering the fact that the neural network represents a compositional function, automatic differentiation applies the chain rule repeatedly to compute the derivatives.In particular, there are two steps in automatic differentiation:

- A forward pass to compute the values of all variables-
- A backward pass to compute the derivatives.

The important feature of automatic differentiation is that it only requires one forward and one backward pass to compute all the derivatives, no matter what the input dimension is. Hence, it is more efficient than finite difference when the input dimension is high.

1.1.3 Algorithm of PINNs

We consider the following PDE parametrized by $\boldsymbol{\lambda}$ for the solution $\boldsymbol{u}(\mathbf{x})$ with $\mathbf{x} = (x_1, ..., x_d)$ defined in a domain Ω

$$f(\mathbf{x}; \frac{\partial \boldsymbol{u}}{\partial x_1}, ..., \frac{\partial \boldsymbol{u}}{\partial x_d}; \frac{\partial^2 \boldsymbol{u}}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 \boldsymbol{u}}{\partial x_1 \partial x_d}; ..., \boldsymbol{\lambda}) = 0 \tag{1.1}$$

with suitable boundary conditions defined on $\delta\Omega$

$$\mathcal{B}(\boldsymbol{u}(\mathbf{x})) = 0$$

where these boundary conditions could be Dirichlet, Neumann, Robin, or periodic boundary conditions. Here, we briefly describe the algorithm for PINNs:

- **STEP 1**: Construct a neural network $\hat{\boldsymbol{u}}(\mathbf{x};\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$;
- **STEP 2**: Specify the two training sets $\mathcal{T}_f$ and $\mathcal{T}_b$ for the equation and boundary/initial conditions.
- **STEP 3**: Specify a loss function by summing the weighted L^2 norm of both the PDE and boundary condition residuals;
- **STEP 4**: Train the neural network to find the best parameters $\boldsymbol{\theta}^*$ by minimizing the loss function.

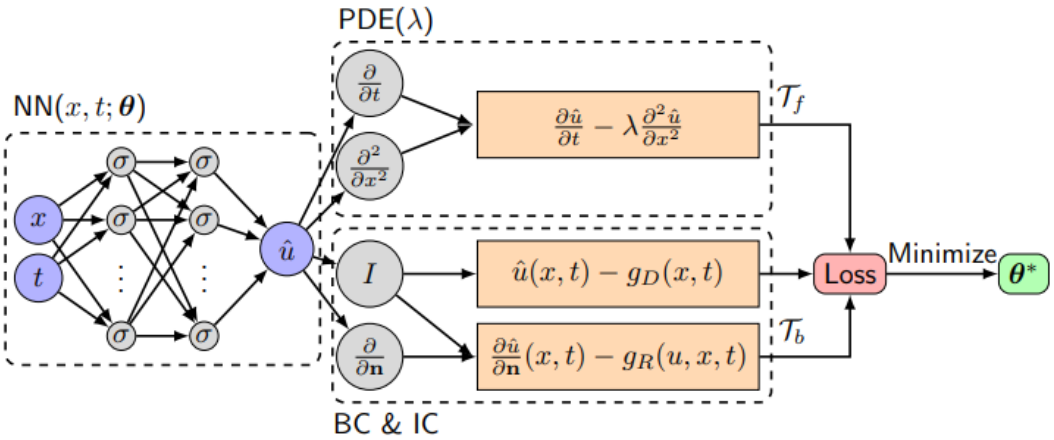


Figure 1.1: Schematic of a PINN for solving the 1D diffusion equation with mixed boundary conditions

To better explain the algorithm above, in a PINN we first construct a neural network $\hat{\boldsymbol{u}}(\mathbf{x};\boldsymbol{\theta})$ as a surrogate of the solution $\boldsymbol{u}(\mathbf{x})$ which takes $\mathbf{x}$ as the input and outputs a vector with the same dimension of $\boldsymbol{u}$. $\boldsymbol{\theta} = (\boldsymbol{W}^l, \boldsymbol{b}^l)_{1 \le l \le L}$ is the set of all weight matrices and bias vectors in the neural network. One advantage of PINNs by choosing neural networks as a surrogate of $\boldsymbol{u}$ is that we can take derivatives of $\hat{\boldsymbol{u}}$ with respect to its input $\mathbf{x}$ by applying the chain rule to differentiate the compositions of functions using automatic differentiation, which is integrated in machine learning packages, such as TensorFlow and PyTorch. In the following step, we must restrict our neural network to satisfy the physics conditions imposed by the PDE and the boundary conditions. We need to restrict $\hat{\boldsymbol{u}}$ to some points (randomly distributed points or clustered points in the domain), i.e. training data $\mathcal{T} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_D)$ of size D. In particular, $\mathcal{T}$ is the union of two different sets $\mathcal{T}_f \subset \Omega$ and $\mathcal{T}_b \subset \delta\Omega$ which are the points in the domain and on the boundary, respectively.
To measure the discrepancy between the neural network $\hat{\boldsymbol{u}}$ and the constraints, we consider the loss function defined as the weighted sum of the L^2 norm of the residual for the equation and the boundary conditions:

$$\mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \omega_f \mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) + \omega_b \mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) \tag{1.2}$$

where

$$\mathcal{L}_f(\boldsymbol{\theta}; \mathcal{T}_f) = \frac{1}{|\mathcal{T}_f|} \sum_{\mathbf{x} \in \mathcal{T}_f} \left\| f(\mathbf{x}; \frac{\partial \hat{\boldsymbol{u}}}{\partial x_1}, ..., \frac{\partial \hat{\boldsymbol{u}}}{\partial x_d}; \frac{\partial^2 \hat{\boldsymbol{u}}}{\partial x_1 \partial x_1}, ..., \frac{\partial^2 \hat{\boldsymbol{u}}}{\partial x_1 \partial x_d}; ..., \boldsymbol{\lambda}) \right\|^2 \tag{1.3}$$

$$\mathcal{L}_b(\boldsymbol{\theta}; \mathcal{T}_b) = \frac{1}{|\mathcal{T}_b|} \sum_{\mathbf{x} \in \mathcal{T}_f} \|\mathcal{B}(\mathbf{x}, \hat{\boldsymbol{u}})\|^2 \tag{1.4}$$

In the last step the training, i.e. the procedure of searching for a good $\boldsymbol{\theta}$ by minimizing the loss, is performed. Since the loss is highly nonlinear and convex with respect to $\boldsymbol{\theta}$, it is usually minimized by a gradient-based optimizer, such as gradient descent, Adam and L-BFGS. For smooth PDE solutions, L-BFGS can find a good solution with less iteration than Adam because it exploits second-order derivatives of the loss function while Adam relies on first-order derivatives. However, for stiff solutions, L-BFGS is more likely to be stuck at a bad local minimum. The required number of iterations depends highly on the problem (e.g., the smoothness of the solution), and to check whether the network converges or not, we can monitor the loss function or the PDE residual using callback functions. It is also possible to note that acceleration of training can be achieved by using an adaptive activation function that may remove bad local minima. Unlike traditional numerical methods, for PINNs there is no guarantee of unique solutions because PINN solutions are obtained by solving non-convex optimization problems, which generally do not have a unique solution. In practice, to achieve a good level of accuracy, we need to tune all the hyperparameters, e.g., network size, learning rate, and the number of residual points. The required network size highly depends on the smoothness of the PDE solution.

1.2. The Random Feature Method

The starting point of this new proposed Random Feature Method is a combination of rather simple ideas: the use of random feature functions to represent the approximate solution, the collocation method to take care of the PDEs and the boundary conditions in the least-square sense, and a rescaling procedure to balance the contributions from the PDEs and the boundary conditions in the loss function. In particular:

- The preferred choices of the basis functions are random feature functions. If necessary, the actual choice of basis functions can be tuned beforehand in a pre-computing stage in order to make them more adapted to the nature of the problem. The feature functions used in this neural network have random but fixed weights and biases for the activation function of the single hidden layer. This means that we are using a special class of random feature models: models that come from a two-layer neural network with fixed inner parameters.
- Another important component is the multiscale representation of the solutions. We use a partition of unity (PoU) to piece together different local representations of the solution.
- A rescaling procedure is needed to balance the contributions from the PDE and the boundary conditions in the loss functions by tuning the weight parameters.

Our objective is to implement this proposed methodology to solve PDEs that shares the merits of classical and machine learning-based algorithms ([1], [2]). This new class can be made spectrally accurate, and, at the same time, they are also mesh-free, making them easy to use even in settings with complex geometry.

2. Mathematical formulation of the problem

Consider the following problem:

$$\begin{cases} \mathcal{L}u(\mathbf{x}) = f(\mathbf{x}) & \mathbf{x} \in \Omega \\ \mathcal{B}u(\mathbf{x}) = g(\mathbf{x}) & \mathbf{x} \in \delta\Omega \end{cases} \tag{2.1}$$

The Random Feature Method relies on three key components:

- The loss funcTion built on the least-squares formulation of the PDEs on collocation points.
- The approximate solution constructed using a set of random feature functions.
- The training step with the additional rescaling of the penalty parameters to balance the contributions from different terms.

2.1. Random Feature Functions

Following the random feature model in machine learning, we construct the approximate solution u_M of u by linear combination of M network basis functions Φ_m over Ω as follows

$$u_M(\mathbf{x}) = \sum_{i=1}^{M} u_m \phi_m(\mathbf{x}) \tag{2.2}$$

For vectorial solutions, we approximate each component of the solution using

$$\boldsymbol{u}_M(\mathbf{x}) = (\sum_{i=1}^{M} u_m^1 \phi_m^1(\mathbf{x}), .., \sum_{i=1}^{M} u_m^{K_l} \phi_m^{K_l}(\mathbf{x}))^T \tag{2.3}$$

The basis functions will be chosen as the ones that occur naturally in neural networks, for example

$$\phi_m(\mathbf{x}) = \sigma(\boldsymbol{k_m} \cdot \mathbf{x} + b_m) \tag{2.4}$$

where σ is some scalar, non linear function (sin, cos, tanh,...), $\boldsymbol{k_m}$ and b_m are some random but fixed parameters. For solving PDE problems, activation functions such as sin, tanh and cos can all be used. Some additional properties could be needed to achieve better performance.

2.2. Loss function

There are three standard approaches to solving (2.1): the weak form, the strong form, and the variational form. Each of these approaches gives rise to some particular choices in the loss function. In this project, we will focus on the strong form at the collocation points to build our loss function. In particular, we have two sets of collocation points: C_I, the set of interior points in Ω and C_B, the set of boundary points of $\delta\Omega$. Let $C = C_I \cup C_B$ be the set of all collocation points. At each collocation point, we will enforce the PDE or the boundary conditions. Let K_I and K_B be the number of conditions at each interior and boundary point. The total number of conditions is $N = K_I \# C_I + K_B \# C_B$. A simple choice of the loss function is the following:

$$Loss = \sum_{\boldsymbol{x}_i \in C_I} \sum_{k=1}^{K_I} \lambda_{I_i}^k \left\| \mathcal{L}^k \hat{\boldsymbol{u}}(\boldsymbol{x}_i) - \boldsymbol{f}^k(\boldsymbol{x}_i) \right\|_{l_2}^2 + \sum_{\boldsymbol{x}_j \in C_B} \sum_{l=1}^{K_B} \lambda_{B_j}^l \left\| \mathcal{B}^l \hat{\boldsymbol{u}}(\boldsymbol{x}_j) - g^l(\boldsymbol{x}_j) \right\|_{l_2}^2 \tag{2.5}$$

This *Loss* is a functional of the approximated solution $\hat{\boldsymbol{u}}$ that must be proper minimized to obtain the best approximation of the true solution of the problem. Here, $\lambda_{I_i}^k$ and $\lambda_{B_j}^l$ are the penalty parameters. In this form, we allow different choices of the penalty parameters at different collocation points. By treating the boundary conditions and the PDE on the same footing, we do not need to impose boundary conditions for the feature function.

2.3. Partition of Unity

Random feature functions are defined globally, while the solution of PDEs typically has local variations, possibly at small scales. To avoid loss of precision, we construct many local solutions, each of which corresponds to a random feature model, and piece them together using partition of unity (PoU) (see [1] and [3]). Consider a partition of unity $\psi = \{\psi_\alpha(\mathbf{x})\}_{\alpha=1}^{N_{part}}$, satisfying $\sum_\alpha \psi_\alpha(\mathbf{x}) = 1$ and $\psi_\alpha \geq 0$ for all $\mathbf{x}$. To construct our PoU, we assume that we have an interval $I = [-1, 1]$ and we consider the definition given in [4], which is:

$$\psi(x) = \begin{cases} \frac{1+sin(2\,\pi\,x)}{2} & -\frac{5}{4} \leq x < -\frac{3}{4} \\ 1 & -\frac{3}{4} \leq x < \frac{3}{4} \\ \frac{1-sin(2\,\pi\,x)}{2} & \frac{3}{4} \leq x < \frac{5}{4} \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

Starting from this definition, we considered a slightly modified construction, where the two sinusoidal parts are approximated by two polynomial functions that have zero derivatives at $\pm\frac{3}{4}$ and $\pm\frac{5}{4}$. This definition is then shifted accordingly with the number of partitions and the domain interval considered. The implementation of this method is explained in Section 3.3.

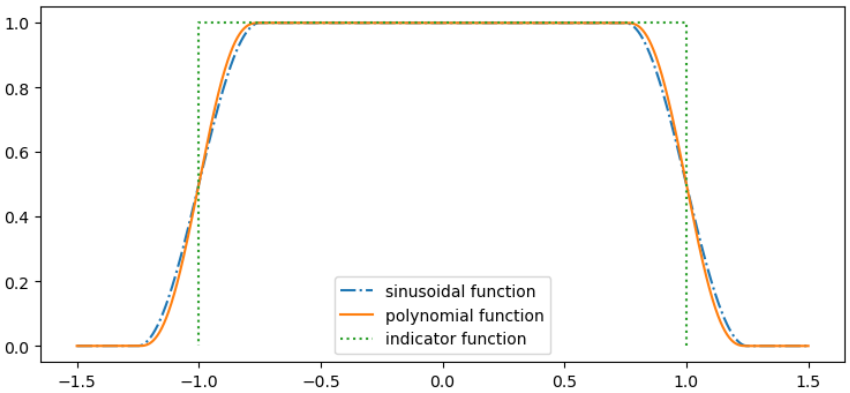


Figure 2.1: Comparison between sinusoidal and polynomial PoU functions

Next, we construct J_n random feature function:

$$\phi_{nj}(\mathbf{x}) = \sigma(\boldsymbol{k_{nj}} \cdot \tilde{\boldsymbol{x}} + b_{nj}) \tag{2.7}$$

where $(\boldsymbol{k_{nj}}, b_{nj})$ are randomly chosen. A common choice is the uniform distribution $\boldsymbol{k_{nj}} \sim \mathcal{U}([-R_{nj},\, R_{nj}]^d)$ and $b_{nj} \sim \mathcal{U}([-R_{nj},\, R_{nj}])$, where the variable R_{nj} depends on the problem.
Putting together, the approximate solution of (2.1) u_M is given by

$$u_M(\mathbf{x}) = \sum_{n=1}^{M_p} \psi_n(\mathbf{x}) \sum_{j=1}^{J_n} u_{nj} \phi_{nj}(\mathbf{x}) \tag{2.8}$$

3. Methods

In the following paragraphs we briefly describe how we implemented the Random Feature Method. In Section 3.1 we introduce the library, used as a starting point for our implementation, while in Sections 3.2 and 3.3 we present our code.

3.1. DeepXDE

DeepXDE is a Python library for PINNs. Specifically, DeepXDE can solve forward problems given initial and boundary conditions, as well as inverse problems given some extra measurements. DeepXDE supports complex-geometry domains based on the technique of constructive solid geometry and enables the user code to be compact, resembling closely the mathematical formulation. By using DeepXDE, time-dependent PDEs can be solved as easily as steady states by only defining the initial conditions. In addition to the main workflow of DeepXDE, users can readily monitor and modify the solution process via callback functions, e.g. monitoring the Fourier spectrum of the neural network solution, which can reveal the learning mode of the NN. DeepXDE supports five tensor libraries as backends: TensorFlow 1.x, TensorFlow 2.x, PyTorch, JAX, and Paddle. For more details, refer to [4]. For this project, Tensorflow 2.x is the back-end chosen for the usage of the library and the implementation of the new method. DeepXDE is a very wide library; in the following sections, we describe the general flowchart to solve PDEs and the main functions used to implement the Random Feature Method.

3.1.1 DeepXDE flowchart

The following procedure briefly describes the basic usage of the library to solve Partial Differential Equations.

Procedure 1 Usage of DeepXDE to solve partial differential equations

Step 1: Specify the computational domain using the **geometry** module
Step 2: Specify the PDE using **Tensorflow** commands
Step 3: Specify BCs and ICs
Step 4: Combine geometry, PDE, and boundary/initial conditions together into **data.PDE** or **data.TimePDE** for time-independent problems or for time-dependent problems, respectively. To specify training data, it is possible to set the specific point locations, or only set the number of points, and then DeepXDE will sample the required number of points on a grid or randomly.
Step 5: Construct a neural network using the **nn** or **fnn** module
Step 6: Define a **Model** by combining Step 4 and 5
Step 7: Call **Model.compile** to set optimization hyperparameters, such as the optimizer and the learning rate. The weights can be set here by loss weights.
Step 8: Call **Model.train** to train the network from random initialization or a pre-trained model using the argument **model restore path**. Use **callbacks** to monitor training behavior.
Step 9: Call **Model.predict** to predict the PDE solution at different locations.

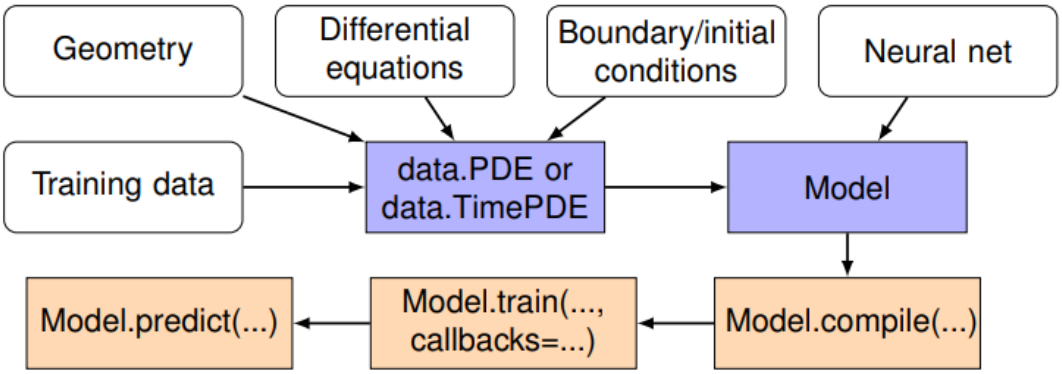


Figure 3.1: Flowchart of the Procedure to solve PDEs

3.1.2 Geometry

DeepXDE has already supported 10 basic geometries for 1D, 2D, 3D and n-dimensional problems, as well as the CSG technique. However, it is still possible for the user to construct new geometry, which is not already implemented in DeepXDE. In particular, for a new geometry the user needs to define all the functions implemented for the other already supported geometries.

3.1.3 Data

The class **data** is an abstract class that is defined using the ABC (Abstract Base Classes) Python module. Since Python does not provide by default abstract classes, it is necessary to use this module and then decorate the methods of the base class as abstract, and then register concrete classes as implementations of the abstract base. On the basis of this abstract class, different classes are implemented.

Pde

In the module **pde** the class defining the PDE to solve is implemented. It takes as arguments:

- the geometry of the problem;
- boundary and initial conditions;
- the definition of the partial differential equation to solve;
- the number of training points sampled inside the domain;
- the number of training points sampled on the boundary;
- the distribution to sample training points. One of the following: "uniform" (equispaced grid), "pseudo (pseudorandom), "LHS" (Latin hypercube sampling), "Halton" (Halton sequence), "Hammersley" (Hammersley sequence), or "Sobol" (Sobol sequence);
- the reference solution;
- the number of points sampled inside the domain for testing PDE loss. The test points for BCs/ICs are the same set of points used for training.

This class constructs the losses regarding the boundary points and the internal points of the considered domain, that have to be optimized during the training step; and then it defines how the training and validation points are sampled, according to the distribution it has given as input.

Fpde

This class allows to define a PDE with d-dimensional fractional laplacian operator to solve. The fractional Laplacian operator is defined as follows:

$$(-\Delta)^{\frac{\alpha}{2}} = \frac{\Gamma(\frac{1-\alpha}{2})\Gamma(\frac{\alpha+D}{2})}{2\pi^{(\frac{D+1}{2})}} \int_{\|\boldsymbol{\theta}\|_2=1} D^{\alpha}_{\boldsymbol{\theta}} u(\mathbf{x}, t) d\boldsymbol{\theta} \tag{3.1}$$

where $1 \leq \alpha \leq 2$, $\boldsymbol{\theta} \in \mathbb{R}^D$ is the differentiation direction vector and $D^{\alpha}_{\boldsymbol{\theta}}$ denotes the directional fractional differential operator. In particular, this solver does not consider $\frac{\Gamma(\frac{1-\alpha}{2})\Gamma(\frac{\alpha+D}{2})}{2\pi^{(\frac{D+1}{2})}}$ in the fractional Laplacian and only discretizes $\int_{\|\boldsymbol{\theta}\|_2=1} D^{\alpha}_{\boldsymbol{\theta}} u(\mathbf{x}, t) d\boldsymbol{\theta}$. $D^{\alpha}_{\boldsymbol{\theta}}$ is approximated by the Grunwald-Letnikov formula. A novel element in solving fractional partial differential equations in the context of PINNs is the hybrid approach that is introduced to construct the residual in the loss function using both automatic differentiation for integer-order operators and numerical discretization for fractional operators. This approach bypasses the difficulties that stem from the fact that automatic differentiation is not applicable to fractional operators because the standard chain rule in integer calculus is not valid in fractional calculus. The numerical discretization for the fractional operator is implemented in the class **Scheme** which discretize fractional Laplacian using the quadrature rule for the integral with respect to the directions and the Grunwald-Letnikov (GL) formula for the Riemann-Liouville directional fractional derivative. For more theoretical and practical details on fractional PINNs refer to [5].

3.1.4 icbc

DeepXDE has already implemented 4 classes defining Dirichlet, Neumann, Robin and periodic boundary conditions. However, it is possible to define a customized boundary condition operator, following the given procedure. It is composed by the **BC** abstract base class and from different classes, one for each type of boundary conditions, that are all based on **BC**.

3.1.5 NN

In the library there are already implemented:

- NN: a base class for all neural network modules;
- FNN: a class for fully-connected neural networks. It takes as arguments the size of the layers, the activation function, the initializer for the kernel, and the dropout rate. DeepXDE also supports PFNN, a class for the definition of parallel fully connected neural network that uses independent subnetworks for each network output. In these classes, the entire neural network architecture is defined, and in the **__call__** function the way the neural network manipulates the inputs given to it.
- Deeponet: a class with a similar definition of the FNN class, but it is designed for dataset in the format of Cartesian product.

3.1.6 Model

In this module all the steps needed to train the neural network are implemented. It trains the neural network defined in 3.1.5 on the pde defined in 3.1.3.
COMPILE It configures the model for training. It takes as arguments:

- the optimizer: string name of an optimizer or a backend optimizer class instance. The most common optimizers are Adam and L-BFGS;
- the learning rate;
- the loss. If the same loss is used for all errors, then it is a string name of a loss function or a loss function. If different errors use different losses, then it is a list whose size is equal to the number of errors.
- the metrics: list of metrics to be evaluated by the model during training;
- the loss weights: a list specifying scalar coefficients (Python floats) to weight the loss contributions. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the loss weight coefficients.

The compile function defines which training and validation losses the neural network should optimize, and it defines the step the neural network should train, using a tensorflow built-in automatic differentiation tool called GradientTape().

```
def _compile_tensorflow(self, lr, loss_fn, decay, loss_weights):
    """tensorflow"""

    @tf.function(jit_compile=config.xla_jit)
    def outputs(training, inputs):
        return self.net(inputs, training=training)

    def outputs_losses(training, inputs, targets, auxiliary_vars, losses_fn)
    :
        self.net.auxiliary_vars = auxiliary_vars

        outputs_ = self.net(inputs, training=training)
        # Data losses
        losses = losses_fn(targets, outputs_, loss_fn, inputs, self)
        if not isinstance(losses, list):
            losses = [losses]
        # Regularization loss
        if self.net.regularizer is not None:
            losses += [tf.math.reduce_sum(self.net.losses)]
        losses = tf.convert_to_tensor(losses)
        # Weighted losses
        if loss_weights is not None:
            losses *= loss_weights
        return outputs_, losses
```

Code 3.1: Model compile: definition of the output losses

```
    opt = optimizers.get(self.opt_name, learning_rate=lr, decay=decay)

    @tf.function(jit_compile=config.xla_jit)
    def train_step(inputs, targets, auxiliary_vars):
        with tf.GradientTape() as tape:
          losses = outputs_losses_train(inputs, targets, auxiliary_vars)[1]
          total_loss = tf.math.reduce_sum(losses)
        trainable_variables = (self.net.trainable_variables + self.
   external_trainable_variables)
        grads = tape.gradient(total_loss, trainable_variables)
        opt.apply_gradients(zip(grads, trainable_variables))

    # Callables
    self.outputs = outputs
    self.outputs_losses_train = outputs_losses_train
    self.outputs_losses_test = outputs_losses_test
    self.train_step = train_step
```

Code 3.2: Model compile: definition of the train step

TRAIN It trains the model. It takes as arguments:

- the number of iterations to train the model;
- the batch size;
- callbacks.

The train step begins when the neural network is supplied with an optimizer during the compiling stage. This initiates the train function. In this module, the prediction function is also defined: It takes as argument the input samples and an operator, which could be set at None. The operator is typically chosen as the pde, so this function computes the residual PDE.

```
633     def _train_sgd(self, iterations, display_every):
634         for i in range(iterations):
635             self.callbacks.on_epoch_begin()
636             self.callbacks.on_batch_begin()
637 
638             self.train_state.set_data_train(
639                 *self.data.train_next_batch(self.batch_size)
640             )
641             self._train_step(
642                 self.train_state.X_train,
643                 self.train_state.y_train,
644                 self.train_state.train_aux_vars,
645             )
646 
647             self.train_state.epoch += 1
648             self.train_state.step += 1
649             if self.train_state.step % display_every == 0 or i + 1 ==
    iterations:
650                 self._test()
651 
652             self.callbacks.on_batch_end()
653             self.callbacks.on_epoch_end()
654 
655             if self.stop_training:
656                 break
```

Code 3.3: Model train: definition of the train function (adam optimizer)

```
707     def _train_tensorflow_tfp(self):
708 
709         n_iter = 0
710         while n_iter < optimizers.LBFGS_options["maxiter"]:
711             self.train_state.set_data_train(
712                 *self.data.train_next_batch(self.batch_size)
713             )
714             results = self.train_step(
715                 self.train_state.X_train,
716                 self.train_state.y_train,
717                 self.train_state.train_aux_vars,
718             )
719             n_iter += results.num_iterations.numpy()
720             self.train_state.epoch += results.num_iterations.numpy()
721             self.train_state.step += results.num_iterations.numpy()
722             self._test()
723 
724             if results.converged or results.failed:
725                 break
```

Code 3.4: Model train: definition of the train function (LBFGS optimizer)

3.2. Implementation of the Random Feature Neural Network

Regarding the structure of random feature neural network, the main difference with respect all the other types of neural network is that we must have only one hidden layer with non-trainable weights and biases. However, in the **random_fnn** class, we implemented a general version of this type of neural network, which could be composed of many hidden layers. The class takes the same inputs of the regular fully connected neural network plus the limits for the non-trainable weights and biases R_m and b such that: $weights \in \mathcal{U}[-R_m; R_m]$ and $biases \in \mathcal{U}[-b; b]$. The random feature layer is generated if in the list of activations, given in input, one layer is identified by the string "random_sin" or "random_tanh". In that case, the weights and biases are randomly and uniformly initialized and the **trainable** input of the Dense layer is set to False. In the variable named **denses** all the dense layers of the network are then listed and this is the key variable used in the **__call__** function where the output of the neural network is computed.

```
25 class random_FNN(NN):
26
27     def __init__(self, layer_sizes, activation, kernel_initializer, Rm=1, b
   =0.0005,
28                  regularization=None, dropout_rate=0):
29
30         super().__init__()
31         self.regularizer = regularizers.get(regularization)
32         self.dropout_rate = dropout_rate
33         self.denses = []
34         fun_activation = list(map(activations.get, activation))
35         initializer = initializers.get(kernel_initializer)
36
37         for j, units in enumerate(layer_sizes[1:-1]):
38
39             if activation[j] == 'random_sin' or activation[j] == '
   random_tanh':
40                 free = False
41                 init = tf.keras.initializers.RandomUniform(minval=-Rm,
   maxval=Rm)
42                 bias = tf.keras.initializers.RandomUniform(minval=-b, maxval
   =b)
43             else:
44                 free = True
45                 init = initializer
46                 bias = "zeros"
47
48             self.denses.append(tf.keras.layers.Dense(units,
49                     activation=(fun_activation[j] if isinstance(
   fun_activation, list)
50                                 else fun_activation),
51                     kernel_initializer=init, kernel_regularizer=self.
   regularizer,
52                     bias_initializer=bias, trainable=free))
53
54         self.denses.append(tf.keras.layers.Dense(layer_sizes[-1],
55                 kernel_initializer=initializer, kernel_regularizer=self.
   regularizer))
```

Code 3.5: initializing the random feature neural network

3.3. Partition of Unity

3.3.1 PoU indicators

As stated in section 2.3, we define our partition of unity function as a polynomial approximation of (2.6). In particular, the functions that approximate the two sides of the sinusoidal PoU are:

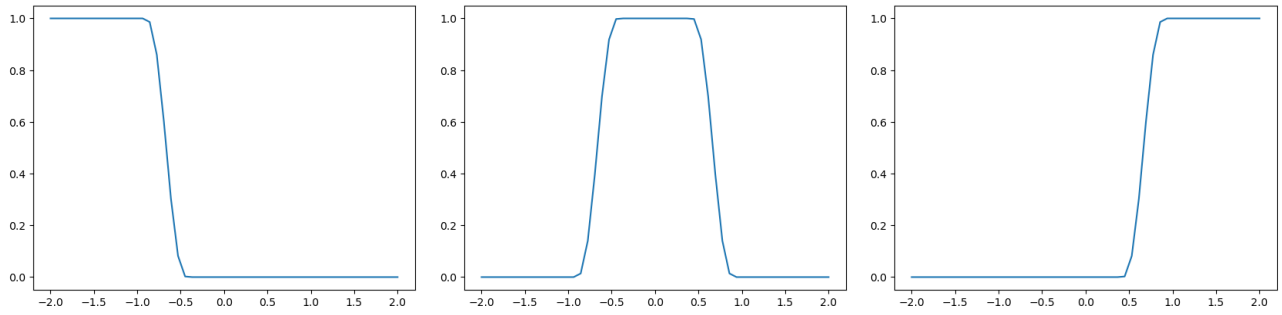
$$\psi(x) = \begin{cases} (\frac{1}{5}x^5 + x^4 + \frac{47}{24}x^3 + \frac{15}{8}x^2 + \frac{225}{256}x + 0.162)\,10^{-3} & -\frac{5}{4} \le x < -\frac{3}{4} \\ 1 & -\frac{3}{4} \le x < \frac{3}{4} \\ (\frac{1}{5}x^5 - x^4 + \frac{47}{24}x^3 - \frac{15}{8}x^2 + \frac{225}{256}x - 0.162)\,10^{-3} & \frac{3}{4} \le x < \frac{5}{4} \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In order to construct the partition of unity in Figure 2.1, we used the tensorflow function **tf.clip_by_value**, which is a function that clips the tensor values to a specified minimum and maximum value. It's preferable to use a tensorflow function to define the PoU rather then a similar NumPy function, to not have compatibility issues while using it inside the neural network **__call__** (see code 3.8). In the code 3.6, we defined the two sides of (3.2) in the functions **func_dx** and **func_sx** and then they are composed together with **tensorflow.clip_by_value** to obtain our partition of unity. One important feature to consider is that, considering any interval, the sum of partition's indicators over every interval's point must be equal to 1. For that reason, we defined 3 types of PoU indicators that depend on which partition we are considering during the construction. Indeed, the first and last partition's indicators would be equal to 1 for all the points near to the boundary, while the other partition's indicators would be equal to 0.5 in the intersection points. An example is presented in Figure 3.2.

```
    def func_dx(x, a=0):
    c1 = - 0.162
    res = (c1 + 1/5*(x-a)**5 - (x-a)**4 + 47/24*(x-a)**3
           - 15/8*(x-a)**2 + 225/256*(x-a))/0.001
    return res

def func_sx(x, a=0):
    c2 =  0.162
    res = (c2 + 1/5*(x-a)**5 + (x-a)**4 + 47/24*(x-a)**3
           + 15/8*(x-a)**2 + 225/256*(x-a))/0.001
    return res

def pou(x, a=-1, b=1):
    return tf.clip_by_value(1-func_dx(x, b-1), 0.0, 1.0)
           + tf.clip_by_value(func_sx(x, a+1), 0.0, 1.0) - 1

def pou_dx(x, b=1, one=1.5):
    return tf.clip_by_value(func_sx(x, b-1+one), 0.0, 1.0)

def pou_sx(x, a=-1, one=1.5):
    return tf.clip_by_value(1-func_dx(x, a+1-one), 0.0, 1.0)

def indicator(lim_sx, lim_dx, a, b, npart, i):
    if i == 0:
        return lambda x: pou_sx(x, a, np.abs(b) + (2 + lim_sx))
    elif i == npart-1:
        return lambda x: pou_dx(x, b, np.abs(a) + (2 - lim_dx))
    else:
        return lambda x: pou(x, a, b)

def pou_indicators(geom, npart):
    lim_dx = geom.r
```

```
    lim_sx = geom.l
    arr = np.linspace(lim_sx, lim_dx, npart+1)
    res = []
    for i in range(npart):
        res.append(indicator(lim_sx, lim_dx, arr[i], arr[i+1], npart, i))
    return res
```

Code 3.6: definition of the PoU indicator functions



(a) first partition (b) second partition (c) third partition

Figure 3.2: PoU indicator functions for the interval $I = [-2, 2]$ divided into 3 partitions

The **partition_random_FNN** class is the one which constructs a neural network composed by as many sub-networks as the number of partitions chosen and it's the neural network that returns as output the approximate solution given by (2.8). The class takes the same inputs as the **random_fnn** class plus 2 more variables, which are the number of partitions considered (npart) and a list containing the PoU function for each partition (nn_ind). The PoU construction (code 3.7) is based on the idea of initializing a number of different **random_fnn** neural networks and then saving all their layers in a single list. In its **__call__** method (code 3.8), we iterate over all the partitions and apply each single neural network to all the inputs. At the end the output is multiplied by the PoU function and then it is summed with the other networks' outputs. This sum is the output of our **partition_random_FNN** neural network.

```
class partition_random_FNN(NN):

    def __init__(self, layer_sizes, activation, kernel_initializer, npart,
                 nn_ind, Rm=1, b=0.0005, regularization=None, dropout_rate
    =0):

        super().__init__()
        self.regularizer = regularizers.get(regularization)
        self.dropout_rate = dropout_rate

        self.nets = [random_FNN(layer_sizes, activation, kernel_initializer,
                    Rm, b, regularization, dropout_rate) for i in range(
    npart)]

        self.denses = [self.nets[i].denses for i in range(npart)]
        self.nn_ind = nn_ind
        self.npart = npart
```

Code 3.7: initializing the partition of unity neural network

```
149 def __call__(self, inputs, training=True):
150 
151         x = inputs
152         res = 0
153         for i in range(self.npart):
154             y = inputs
155             indicator = self.nn_ind[i](x)
156 
157             if self._input_transform is not None:
158                 y = self._input_transform(y)
159 
160             for f in self.denses[i]:
161                 y = f(y, training=training)
162 
163             if self._output_transform is not None:
164                 y = self._output_transform(inputs, y)
165 
166             y = tf.math.multiply(y, indicator)
167             res += y
168         return res
```

Code 3.8: **__call__()** function for the partition of unity neural network

4. Numerical results

In this section we show the most significant numerical results obtained by adopting the implemented Random Feature Method. In particular, this new proposed methodology has been tested to solve different PDEs problems. We consider problems with explicit solutions to study how the performance of the RFM depends on the different components in the algorithm. Indeed we performed different sensitivity analysis on the parameters that play a significant role in the model to better understand how to improve the performance of the method. A comparison with the performances of a classical PINNs approach is also shown to highlight the advantages of the Random Feature Method.
We briefly recall which are the main parameters we focused on:

- M = it's the hidden layer dimension; so it's the number of random basis function we are considering
- k_m, b_m = they are the random but fixed weights and biases representing the activation function of the hidden layer. They are usually sampled from a uniform distribution.
- R_m = it's the coefficient that describes the domain of the uniform distributions from which k_m and b_m are sampled

4.1. Parameters sensitivity analysis for the Random Feature Method and comparison with PINNs

Problem 4.1. *Consider the one-dimensional Helmhotz equation with Dirichlet boundary conditions over the domain* $\Omega = [-3, 2]$

$$\begin{cases} -\frac{d^2u(x)}{dx^2} + u(x) = f(x) & x \in \Omega \\ u(-1) = c_1, \quad u(1) = c_2 \end{cases} \tag{4.1}$$

The explicit form of the solution u *is assumed to be* $u(x) = 0.5\,x^5 + 1.3\,x^4 - 2.7\,x^3 - 5.5\,x^2 + 2.7\,x + 2.3$. *Once the explicit form is given,* c_1, c_2 *and* $f(x)$ *can be computed accordingly.*

In this problem we set the hyperparameters as follows:

- $M = 25,\ 50,\ 100,\ 150,\ 200,\ 250$
- number of training points $n_p = 40$
- $R_m = 2,\ 4,\ 6,\ 8,\ 10$
- weights $k_m \sim \mathcal{U}([-R_m,\ R_m])$
- biases $b_m \sim \mathcal{U}([-R_m,\ R_m])$

According to Table 4.1 different considerations can be exploited.
Fixing the value of R_m it is possible to observe a common trend: when increasing the number of basis functions used to approximate the solution, the accuracy of the solution increases. This trend reflects the expectations on this comparison: increasing the complexity of the neural network will guarantee a better approximation of the solution up to some overfitting problems that need to be taken into consideration. The satisfying result is that this new implemented method performs even with a limited number of basis functions.
On the contrary, fixing the value of M it is possible to observe an interesting behavior: the network seems to perform better when $R_m = 4,\ 6$. From our understanding of the problem, the value of R_m is strictly related to the frequency of the approximated solution. Values of R_m smaller than the highest frequency of the exact solution will not be able to catch the exact behavior of the solution. On the contrary, using values of R_m much bigger than the frequency would need a more complex neural network, and as a consequence more training data, to well approximate the solution. In conclusion, some a priori spectral knowledge on the exact solution would definitely help in reaching the best performance of the Random Feature Method.
For what concerns the comparison between the RFM and a classical PINNs approach, we generally obtain the same performance in terms of the accuracy of the approximated solution. The main advantage of this new method is the significantly reduced computational cost, as is possible to observe

from the number of epochs needed to reach the optimal approximation. Indeed, in RFM, the number of network parameters to optimize is $M + 1$; while in the classic PINNs method, the number of network parameters to optimize is $2\,M + (M + 1)$. This leads to a great increase in complexity when we increase the number of basis functions. The bad result obtained in the case of PINNs with $M = 250$ is only due to the fact that we are considering a small number of training points with respect to the complexity of the network, so we would need to increase that number to obtain a satisfactory result. Otherwise, the RFM still shows good performance without the need of increasing the constraints.

M	R_m	RFM loss	RFM epochs	PINN loss	PINN epochs
25	2	1.820e-01	17	2.438e-01	904
	4	2.759e-01	22		
	6	2.032e-02	22		
	8	1.205e-01	22		
	10	4.840	25		
50	2	6.928e-01	13	5.787e-02	1313
	4	6.293e-03	19		
	6	2.623e-02	26		
	8	1.302e-02	25		
	10	9.169e-03	33		
100	2	6.045e-02	12	8.747e-03	1494
	4	1.639e-03	17		
	6	5.557e-03	23		
	8	2.602e-02	32		
	10	8.599e-03	33		
150	2	2.758e-02	12	1.305e-02	814
	4	2.182e-03	20		
	6	1.218e-02	22		
	8	2.268e-02	28		
	10	1.095e-02	29		
200	2	1.099e-02	15	5.986e-02	1284
	4	1.863e-03	19		
	6	1.776e-03	22		
	8	2.199e-02	28		
	10	1.145e-02	32		
250	2	1.604e-02	14	836.355	8
	4	2.048e-03	20		
	6	2.370e-03	23		
	8	2.092e-02	27		
	10	2.030e-02	33		

Table 4.1: L-BFGS optimizer: Comparison between Random Feature Method and classic PINN method validation losses

M	R_m	Adam loss	L-BFGS loss
50	4	6.293e-03	7.302e-03
	6	2.623e-02	5.117e-03
	8	1.302e-02	7.262e-02
200	4	1.863e-03	4.021e-03
	6	1.776e-03	1.710e-03
	8	2.092e-02	6.876e-02

Table 4.2: Adam vs L-BFGS: validation losses comparison

Regarding the comparison between the Adam and L-BFGS optimizers presented in Table 4.2, we observe that both the optimizers show generally the same performances in terms order of accuracy. However, it is worth underlining that the computational cost of the overall process is significantly reduced using L-BFGS as an optimizer. A single L-BFGS iteration is much more computationally expensive than a single Adams iteration; on the contrary, the L-BFGS method reach an optimal solution in far fewer iterations. This leads to the fact that the computational gain in using L-BFGS is significant. In fact, to obtain the results presented in Table 4.2, the Adam ones were obtained in 50 seconds, while the L-BFGS ones in 5 seconds.

4.2. Difference between single random feature method and Partition of Unity method

Problem 4.2. *Consider the one-dimensional Poisson equation with Dirichlet boundary conditions over the domain* $\Omega = [-2, 2]$

$$\begin{cases} -\frac{d^2u(x)}{dx^2} = f(x) & x \in \Omega \\ u(-1) = c_1, \quad u(1) = c_2 \end{cases} \tag{4.2}$$

The explicit form of the solution u *is assumed to be* $u(x) = sin(3\,\pi\,x + \frac{3\,\pi}{20}) * cos(2\,\pi\,x + \frac{\pi}{20}) + 2$. *Once the explicit form is given,* c_1, c_2 *and* $f(x)$ *can be computed accordingly.*

In this problem, we set the hyperparameters as follows:

- number of partitions $N = 2,\ 3,\ 4,\ 5,\ 6$;
- $M = \frac{1200}{N}$
- number of training points $n_p = 100$ (Table 4.3) and $n_p = 30\,N + 30$ (Table 4.4)
- $R_m = 20$ for each partition $n = 1,\ ...,\ N$
- weights $k_m \sim \mathcal{U}([-R_m,\ R_m])$
- biases $b_m \sim \mathcal{U}([\frac{-R_m}{10},\ \frac{R_m}{10}])$

In the single random feature model we consider a fully-connected neural network, composed by a hidden layer with 1200 neurons and the *sin* activation function is used. On the contrary, in the PoU model we consider N neural networks, composed of a hidden layer with M neurons and *sin* activation function is used. The error of the different reconstructions and some of the solution reconstructions are presented in Table 4.3 and Figure 4.1. It is clear that the performance of the method significantly improves as the number of partitions used to reconstruct the solution. It is also worth underlining that to obtain good performance further increasing the number of partitions, there is also a need to increase the number of training points to have enough of them to not underfit each local neural network, so to properly reconstruct each local solution. This is why with 5 partitions an inverse trend of accuracy of the reconstructed solution is observed; increasing the number of training points we recover a better performance of the method.

n partitions	Validation Loss
2	7.515e-01
3	4.549e-01
4	5.977e-02
5	2.315e-01
6	2.219e-01

Table 4.3: Comparison of the different number of partitions for the problem 4.2, $n_p = 100$

n partitions	Validation Loss
2	8.711e-02
3	6.383e-02
4	4.091e-02
5	1.247e-01
6	1.131e-01

Table 4.4: Comparison of the different number of partitions for the problem 4.2, $n_p = 30\,N + 30$

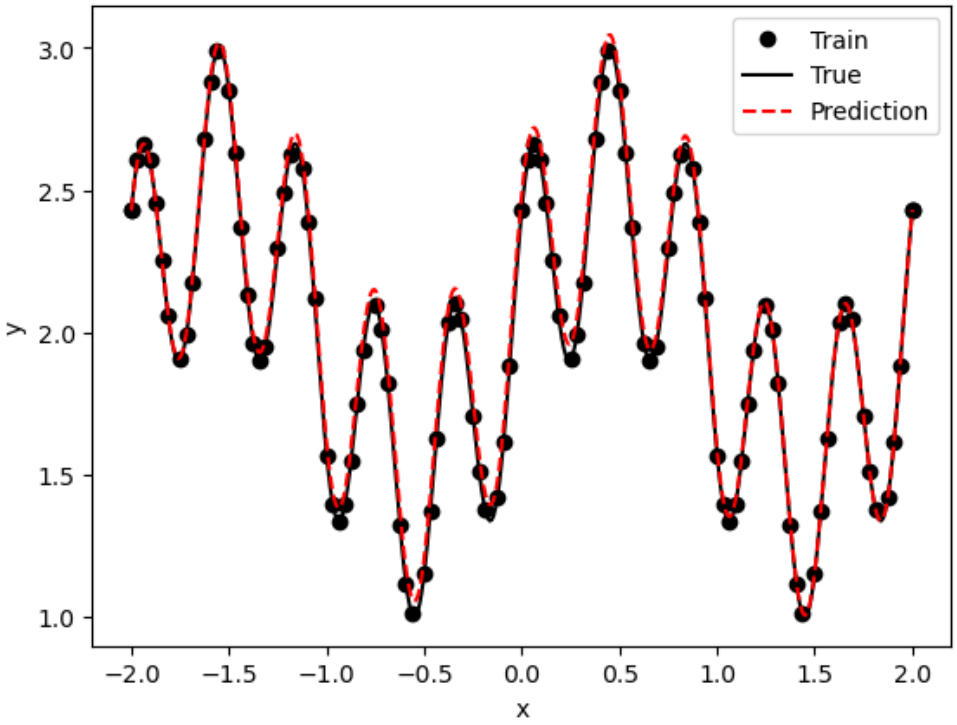


Figure 4.1: Approximated solution vs exact solution using 4 partitions

4.3. Numerical results for fractional PDEs

Problem 4.3. *We consider the time-dependent, one-dimensional fractional Poisson equation, defined over the domain* $\Omega = [0, 1] \times [0, 1]$

$$\begin{cases} \frac{\partial u(x,t)}{\partial t} + (-\Delta)^{\frac{\alpha}{2}} u(x, t) = f(x, t) & \forall (x, t) \in \Omega \\ u(0, t) = u(1, t) = 0 & \forall t \in [0, 1] \\ u(x, 0) = x^3(1 - x)^3 & \forall x \in [0, 1] \end{cases} \tag{4.3}$$

The explicit form of the solution u *is* $u(x, t) = e^{-t}\, x^3\, (1 - x^3)$*, while the parameter* α *is taken as* $\alpha = 1.8$*, according to Section 3.1.3.*

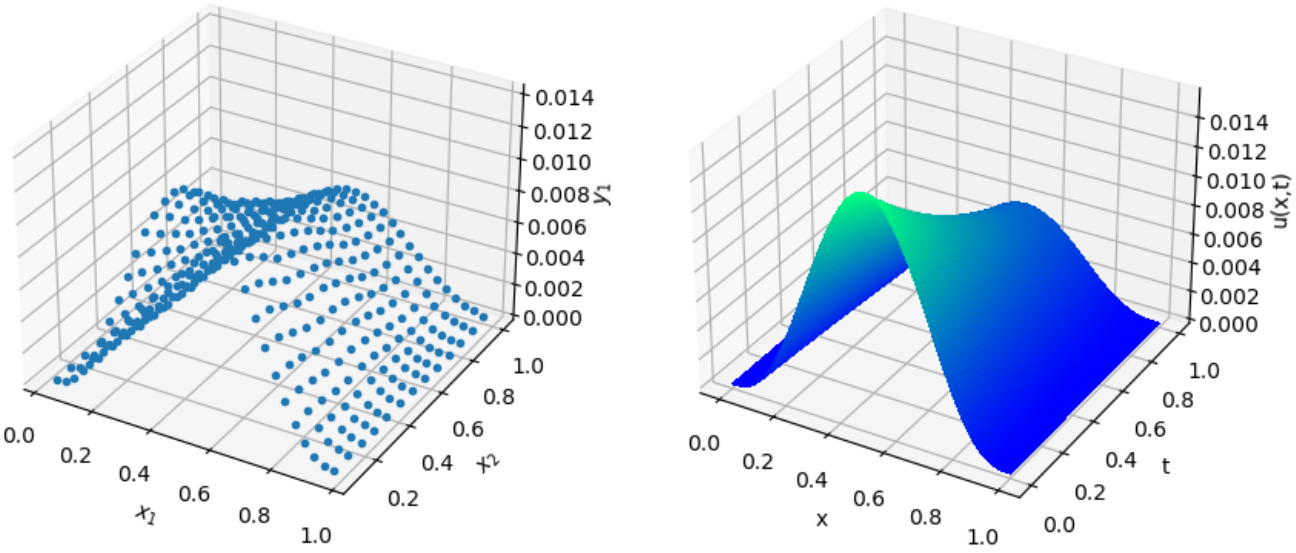
In this problem, we set the hyperparameters as follows:

- $M = 10,\ 25,\ 50,\ 100$
- $R_m = 2,\ 4,\ 5,\ 6,\ 8$
- number of training points $n_p = 50$
- weights $k_m \sim \mathcal{U}([-R_m, R_m])$
- biases $b_m \sim \mathcal{U}([\frac{-R_m}{10}, \frac{R_m}{10}])$

This example allows us to show that the RFM performs well even in the case of a fractional time dependent partial differential equation. For what concerns the performances, the results obtained are similar to the ones analysed in section 4.1. The Random Feature Method outperforms the fPINNs approach when fixing the neural network complexity.

M	R_m	Validation Loss
10	1	3.576e-03
	2	3.342e-03
	4	4.971e-03
	5	4.146e-03
	6	1.032e-03
25	1	6.582e-05
	2	5.946e-05
	4	6.707e-04
	5	6.111e-04
	6	8.002e-04
50	1	3.417e-05
	2	2.153e-05
	4	6.549e-05
	5	1.129e-04
	6	2.383e-04

Table 4.5: Validation loss for problem 4.3 for different values of M and R_m



(a) reconstructed solution

(b) exact solution

Figure 4.2: Approximated solution vs exact solution for problem 4.3

4.4. Numerical results for 2D problem with complex domain

Problem 4.4. *Consider the two-dimensional Poisson equation with Dirichlet boundary conditions over an L-shaped domain* Ω*, defined by its vertices:* $[0,0]$, $[1,0]$, $[1,-1]$, $[-1,-1]$, $[-1,1]$, $[0,1]$

$$\begin{cases} -\Delta u(x,\, y) = 1 & (x,\, y) \in \Omega \\ u(x,\, y) = 0 & (x,\, y) \in \partial\Omega \end{cases} \tag{4.4}$$

The explicit form of the solution $u(x,\, y)$ *is not given.*

In this problem we set the hyperparameters as follows:

- $M = 10,\, 25,\, 50,\, 100$
- Number of training points $n_p = 1200$ or $n_p = 1000 + 15\,M$
- $R_m = 5$.
- weights $k_m \sim \mathcal{U}([-R_m,\, R_m])$
- biases $b_m \sim \mathcal{U}([-R_m,\, R_m])$

In this case we presented a simple two-dimensional PDE but defined over a complex domain. As we can see from Table 4.6 and 4.7 the performances obtained in terms of accuracy are pretty satisfying, nevertheless in this scenario we cannot observe a great accuracy improvement increasing the number of basis function.

M	Validation loss
25	3.168e-04
50	3.129e-04
75	2.342e-04
100	2.403e-04
200	2.433e-04

Table 4.6: Results obtained with $n_p = 1200$

M	Validation loss
25	2.557e-04
50	3.038e-04
75	2.324e-04
100	1.992e-04
200	2.464e-04

Table 4.7: Results obtained with $n_p = 1000 + 15\,M$

5. Conclusions

The main disadvantage of traditional methods to solve PDEs is that they are not flexible enough: most of them require the number of free parameters to be the same as the number of conditions and often require that the basis functions satisfy determined boundary conditions. Compared to traditional algorithms, an important advantage is that the Random Feature Method typically works in a situation where the number of unknown parameters is different from the number of conditions. This forces to use a least-square approach, which results in increasing the complexity of the training process while obtaining reasonable solutions with much lower human and computer cost. Another important difference compared to traditional algorithms is that the boundary conditions are treated in the same way as the PDE. In RFM there is no need to force the basis functions to satisfy particular boundary conditions. This increases significantly the flexibility of the method, making it successful even for problems with complex geometry. Another important feature is the adoption of neural network basis functions. This means that the number of terms needed does not necessarily increase like n^d, where n is the number of unknown parameters in each dimension and d is the dimension. Instead, the number of terms needed depends completely on the complexity of the solution. On the contrary, neural-network-based approach as PINNs typically lacks robustness: it is quite often easy to get roughly good approximations of the solution, but it is very hard to improve the accuracy. The biggest difference compared to PINNs is that RFM uses only the random feature model instead of the neural network model to represent the solution.
According to [1], the PoU random feature method should show an exponential convergence rate for different numbers of partitions. This would improve significantly the performances of the method when compared to a classical PINNs approach, when the same neural-network structure is used, which on the contrary does not show further improvements in the approximation accuracy. However, from our analysis presented in Sections 4.1 and 4.2, both the random feature method and the classic PINNs one show a similar order of accuracy. However, an important advantage that is not presented in [1] is that the new method proposed shows a great improvement on the computational cost side (Table 4.1). This suggests that fixing the inner parameters greatly simplifies the optimization problem and allows us to obtain accurate and robust solutions.
One key aspect for the success of the random feature method is the probability distribution of the feature vector. In practice, we found that, to obtain stable results with the random feature method, the support of the distribution should approximately cover the frequency range of the true solution. We stressed the method over different PDEs, even considering 1D and 2D time-dependent problems and fractional PDEs. According to our analysis, the random feature method seems to be flexible enough to perform well in all these different situations.

6. Future developments

The next steps in analyzing the performance of this new method could be:

- Implement the combination of local and global random feature functions that could allow to obtain better performances in case when the solution has both significant low and high frequency components.
- Understand deeper how the a priori information on the solution helps us to select better random feature functions.
- Stress the method testing it to solve problems with a more complex geometry.
- Analyse the possibility of introducing a rescaling procedure for loss weights to obtain a more accurate approximation of the true solution.
- Analyse the influence of the choice of the collocation points on the accuracy of the solution.

7. Appendix

7.1. Installation

The DeepXDE library with our methods implemented is available at the GitHUb page:

https://github.com/ZIB012/raNNdom.git

Our implementation requires TensorFlow 2.x as backend. Other dependencies that need to run the code are:

- NumPy
- Matplotlib
- scikit-learn
- scikit-optimize
- SciPy

7.2. Use of the library

In this appendix we list the steps to perform in order to use correctly the library and, in particular, the methods we implemented. The code shown below is used to solve the problem 4.1.

- Define the solution domain, through the use of **deepxde.geometry**.
- Define the problem specific pde
- Define the boundary conditions and, if possible, the exact solution of the problem.
- Define the **data** structure in which the training and validation points are constructed.
- Depending on the problem, choose **random_fnn** or **partition_random_fnn** network. If you want to use the PoU version, construct also the partition functions using **deepxde.nn.pou_indicators**. Define the layer size, the activation function, and, eventually, the number of partitions.
- Construct the model, compile, and train it.
- Visualize the reconstructed solution, using the **deepxde.saveplot** function

```
import deepxde as dde
import matplotlib.pyplot as plt
import numpy as np
# Import tf if using backend tensorflow.compat.v1 or tensorflow
from deepxde.backend import tf

geom = dde.geometry.Interval(-3, 2)
def pde(x, y):
    dy_xx = dde.grad.hessian(y, x)
    return -dy_xx + 10*x**3 + 15.6*x**2 - 16.2*x - 11

def boundary(x, on_boundary, npart=1):
    return on_boundary

def func(x):
    return 0.5*x**5 + 1.3*x**4 - 2.7*x**3 - 5.5*x**2 + 2.7*x + 2.3

bc = dde.icbc.DirichletBC(geom, func, boundary)
data = dde.data.PDE(geom, pde, bc, 36, 2, solution=func, num_test=100)

M = 100
layer_size = [1] + [M]  +  [1]

activation = ["random_sin", 'linear']
```

```
25
26 initializer = "Glorot uniform"
27 R = 10
28 net = dde.nn.random_FNN(layer_size, activation, initializer, Rm=R, b=R)
29
30 model = dde.Model(data, net)
31
32 model.compile("adam", lr=0.001, metrics=["l2 relative error"], loss_weights
       =1)
33
34 losshistory, train_state = model.train(iterations=10000)
35
36 dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

Code 7.1: Section 4.1 source code

References

[1] Chen Jingrun et al. "Bridging Traditional and Machine Learning-based Algorithm for solving PDEs: The Random Feature Method". In: *Journal of Machine Learning* (2022).

[2] Jingrun Chen, Weinan E, and Yixin Luo. *The Random Feature Method for Time-dependent Problems*. 2023. arXiv: 2304.06913 [math.NA].

[3] Lee Kookjin et al. "Partition of unity networks: deep hp-approximation". In: (2018).

[4] Lu Lu et al. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: 10.1137/19M1274067.

[5] Pang Guofei, Lu Lu, and George Em Karniadakis. "fPINNs: Fractional Physics-Informed Neural Networks". In: (2018).