BPINN-Eikonal-Inverse-UQ:
Uncertainty quantification of conduction velocities from noisy activation maps using Bayesian Physics-Informed Neural Networks

Author: **Daniele Ceccarelli**
Supervisor: Prof. Andrea Manzoni
Co-supervisor: Dr. Stefano Pagani

Mathematical Engineering, Politecnico di Milano
ADVANCED PROGRAMMING FOR SCIENTIFIC COMPUTING
Prof. Luca Formaggia

A.A. 2019/20

Abstract

In this project we address a computational framework to deal with Inverse Uncertainty Quantification problems related to partial differential equations. To achieve this, we rely on a novel technique in Scientific computing, Physics-Informed Neural Networks (PINN) [9], which merge a data-driven approach with the knowledge of physical based models. In particular we use a variant of PINN within a Bayesian framework, called Bayesian Physics-Informed Neural Network (BPINN) [13]; starting from some prior knowledge, we build a posterior distribution of our parameters (thanks to Bayesian methods like Hamiltonian Monte Carlo or Stein Variational Gradient Descent) and quantify the uncertainty on the computed estimates by reliability intervals.
In particular, we focus on the Eikonal equation, used to describe the electrical activation in heart, in particular the activation times of the tissues. Our goal is to estimate the posterior distribution of the conduction velocities (in order to compute mean and standard deviation) by using data related to sparse and noisy activation time measurements. In this respect, we carry out some experiments relying on synthetic datasets treating both one- and two-dimensional problems set in simple domains and assess the accuracy of the implemented technique and its computational performances. In addition to these examples, also an experiment on a three-dimensional case, on a simple geometry, is shown.

Contents

1 Introduction

1.1 Cardiac Electrophysiology

Clinical procedures for the treatment of electrical disorder phenomena usually relies on a few sparse noisy data of electro-anatomic activation maps, recorded on the cardiac surface using electrodes. These data are processed with deterministic interpolation methods to create an anatomical activation times map of all the cardiac surface. From this measures we are interested in compute the conductivity velocity tensor of the tissue, to see if we can find some area of "small conduction velocity" that can be possibly related to cardiac fibrillation problems.
To obtain these conduction velocity maps, we have to rely on the activation times map, using data affected by noise. In addition to this, the classical methods developed so far do not take into account the physics behind the phenomena, that is a propagation of electric signal in a anisotropic medium, but only the noisy data that we can handle. This is a standard data-driven approach, that usually works well when we have a lot of data, while in this case we have just some sparse observations.
On the other hand, looking at the physical modeling approach, the model that better describe the electrical activity in heart is the so-called Bidomain model, while there are some other simplified model like the Monodomain model and Eikonal model that are simpler but still quite accurate [8].
In this setting the forward model consists of the Eikonal equation, providing the activation time in the domain, and receiving as input the conductivity tensor. On the other hand, the inverse problem - much harder to solved - aims at reconstructing, starting from a set of measured activation times, the conductivity tensor.

1.2 Uncertainty Quantification

In this project we implement methods to solve Inverse Uncertainty Quantification problems of parameter estimation and data assimilation in a Bayesian framework. We rely on Bayesian methods since we are interested in approximating the posterior distributions of the unknown input parameters. Classical methods involving PDE models have some limitations in this class of problems: we have to solve multiple times the PDE to effectively estimate a parametric field with a Bayesian approach.
We adopt a method that avoid the repeated solution of PDEs, however directly providing the posterior distribution of the parameters to be estimated. For this reason, following the works in [9],[10],[13], [12] we have decided to implement a Bayesian Physics-Informed Neural Networks library, that extends the concept of Physics-Informed Neural Networks in a Bayesian framework to solve the Inverse UQ problems for a PDE. The implementation is designed for the Eikonal model but can be extended to any other PDE with minor changes.
In the particular application that we considered, namely cardiac electrophysiol-

ogy, providing an approximation of the posterior distribution of the parametric field we want to estimate, conduction velocities in this case, is a key task, since we start from a few scattered measurements of activation time, mostly affected by noise.
This report is organized as follows: in Section 2 we have described the Eikonal model we focus on. In Section 3 we have described the methods to solve the Inverse UQ problems, namely Bayesian Physics-Informed Neural Networks. In Section 4 we have described the two Bayesian methods used. In Section 5 a very short list of all the external libraries used in the project. In Section 6 we have described actually the code implemented in the project and in Section 7 we have reported some numerical experiments. Finally, our conclusions follow in the Section 8. Some additional details on Tensorflow, graph execution, GPU computation and automatic differentiation are provided in the Appendix.

2 Problem Statement

We focus on the solution of inverse problems related to Eikonal equation.

2.1 Eikonal equation

Let $\Omega \in \mathbb{R}^d$, $d = 1, 2, 3$, be a compact domain, and denote by $T : \mathbb{R}^d \to \mathbb{R}$ the activation time and $\mathbf{M} : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ conductivity tensor (symmetric and positive-defined). The Eikonal equation reads as:
Given the conductivity tensor $\mathbf{M}(\mathbf{x})$ and the localization of the sources $\{\hat{\mathbf{x}}_i\}_{i=1}^{N_S}$, find $T(\mathbf{x})$ such that:

$$\begin{cases} \sqrt{\nabla T(\mathbf{x})^T \mathbf{M}(\mathbf{x}) \nabla T(\mathbf{x})} = 1 & in\ \ \Omega, \\ T(\hat{\mathbf{x}}_i) = 0 \quad \forall i = 1, \dots, N_S. \end{cases} \tag{1}$$

In the forward problem above, given the conductivity tensor $\mathbf{M}(\mathbf{x})$ and the localization of the sources we have to find the activation times map $T(\mathbf{x})$. In the Inverse problem (which is our final goal) we are given the activation times map $T(\mathbf{x})$ in some locations x_j, $j = 1, \dots, N$ (collected by sensors and potentially affected by noise) and the aim is to reconstruct all the entries of the conductivity tensor $\mathbf{M}(\mathbf{x})$.
Accordingly to the form of the conductivity tensor (in dimension $d > 1$), we can distinguish two different forms of the Eikonal equation, the Anisotropic and the Isotropic one.

2.1.1 Anisotropic Eikonal equation

Since the conductivity tensor must be symmetric and positive definite, we can rewrite it in this form for each dimension:

$$\mathbf{M}(\mathbf{x}) = \begin{bmatrix} a(x) \end{bmatrix}, \ \ if\ d = 1$$

$$\mathbf{M}(\mathbf{x}) = \begin{bmatrix} a(x,y) & -c(x,y) \\ -c(x,y) & b(x,y) \end{bmatrix}, \ \ if\ d = 2 \tag{2}$$

$$\mathbf{M}(\mathbf{x}) = \begin{bmatrix} a(x,y,z) & -d(x,y,z) & -e(x,y,z) \\ -d(x,y,z) & b(x,y,z) & -f(x,y,z) \\ -e(x,y,z) & -f(x,y,z) & c(x,y,z) \end{bmatrix}, \ \ if\ d = 3$$

Given this particular reparametrization of $\mathbf{M}(\mathbf{x})$, we can rewrite the equation (1) (for instance in the case $d = 2$) as:

$$\begin{cases} \sqrt{a\left(\frac{\partial T}{\partial x}\right)^2 - 2c\left(\frac{\partial T}{\partial x}\right)\left(\frac{\partial T}{\partial y}\right) + b\left(\frac{\partial T}{\partial y}\right)^2} = 1 & in\ \ \Omega \\ T(\hat{\mathbf{x}}_i) = 0 \quad \forall i = 1, \dots, N_S. \end{cases} \tag{3}$$

2.1.2 Isotropic Eikonal equation

In the particular case where $\mathbf{M}(\mathbf{x}) = v(\mathbf{x})^2 \mathbb{I}_d$, for a suitable function $v : \mathbb{R}^d \to \mathbb{R}$, equation (1) becomes:

$$\begin{cases} \|\nabla T(\mathbf{x})\| = \dfrac{1}{v(\mathbf{x})} & in\ \ \Omega, \\ T(\hat{\mathbf{x}}_i) = 0 & \forall i = 1, \dots, N_S, \end{cases} \tag{4}$$

and is called Isotropic Eikonal equation, since the conduction velocity is the same along all the directions (note that if $d = 1$ the two cases coincide).

3 Physics-Informed Neural Network

Physics-Informed Neural Networks are a new class of powerful methods introduced by Raissi, Karniadiakis and Perdikaris in [9] to solve both forward and inverse problems related to PDEs using neural networks. The idea behind Physics-Informed Neural Network (from now on, PINN) is to create a bridge between data-driven modeling and physics-driven modeling.
Let us consider a generic stationary, nonlinear PDE, under the (abstract) form:

$$\mathcal{N}[u; \lambda] = 0, \quad \mathbf{x} \in \Omega, \tag{5}$$

where $u = u(\mathbf{x})$ is the solution, $\mathcal{N}$ is a non-linear operator and λ is a parametric field. Thanks to PINNs, we can solve both forward problems, whose final goal is to find the solution u given λ, and inverse problems, whose goal is to reconstruct λ from a set of (noisy) measurements of u. In this work we will focus on this latter class of problems.

3.1 PINN for Inverse Problem

Regarding the inverse problem (which is referred to as data-driven *discovery* of PDEs in the PINN framework) we suppose we suppose to know some sparse and noisy data of the exact solution u($\mathbf{x}$) and we want to reconstruct the exact parametric field $\lambda = \lambda(\mathbf{x})$, plus the whole exact solution u($\mathbf{x}$).
Let's consider for example the 2D-Isotropic Eikonal equation:

$$\begin{cases} \|\nabla T(x, y)\| = \dfrac{1}{v(x, y)} & in\ \ \Omega = (0, 1)^2, \\ T(0, 0) = 0 \end{cases} \tag{6}$$

where $T(x, y)$ is the activation times and $v(x, y)$ is the conduction velocity. In this framework our exact solution $u(\mathbf{x})$ is the activation time $T(\mathbf{x})$, while our parameter $\lambda = \lambda(\mathbf{x})$ is the conduction velocity $v(\mathbf{x})$.

After a forward pass in our network, we collect the prediction of T and v as our outputs and then we can use the automatic differentiation to compute all the derivatives required, in this case $\frac{\partial T}{\partial x}$ and $\frac{\partial T}{\partial y}$. Automatic differentiation in

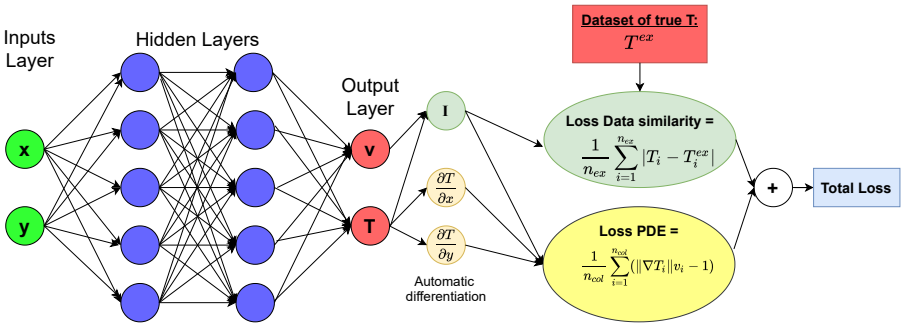


Figure 1: Example of PINN architecture for 2D Isotropic Eikonal

neural networks is a powerful method to compute derivatives: we can compute the **exact** derivative of the output given the input, instead of a numerical approximation as in every numerical differentiation. This is possible because in a neural network the relation that goes from output to input is made by the compositions of an activation function and a linear combination using weight matrix and bias vector: we can compute the true derivative of all these passages (if we suppose to know the activation function derivative, that is true for classical choice of it). After computing the gradient of T for each collocation points, we end up with our physics-informed neural network loss, that is made adding the data-driven loss and physics-driven loss, plus other classical regularization terms we can use. If we denote by $\boldsymbol{\theta}$ all the parameters in our neural network (matrix weights $\mathbf{W}$ and bias vector $\mathbf{b}$ of all the layers L), introduce three additional weights α_{data}, α_{pde} and α_{reg}, we can define our loss as:

$$\mathcal{L}(\boldsymbol{\theta}) = \alpha_d \frac{1}{N_{ex}} \sum_{i=1}^{N_{ex}} (T_i^{\boldsymbol{\theta}} - T_i^{ex})^2 + \alpha_p \frac{1}{N_{col}} \sum_{i=1}^{N_{col}} (\|\nabla T_i^{\boldsymbol{\theta}}\| v_i^{\boldsymbol{\theta}} - 1)^2 + \alpha_r \mathcal{L}^2_{regular}. \quad (7)$$

Solving the PDE and matching the measured data can then be done at the same time, buy minimizing the loss function, using a suitable numerical optimization procedure (such as Gradient Descent or Adam), to find:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}). \quad (8)$$

3.2 Uncertainty Quantification in a Bayesian framework

When we are dealing with real-world data, affected by noise, we are interested also in estimate how likely are our model outcomes: we can't provide just a point evaluation, we need also an estimate of uncertainty on that prediction, for instance approximation its distribution and then compute an interval estimation. We need to compute the posterior distribution of our estimates (and parameters $\boldsymbol{\theta}$), that provides us a bigger and complete information on the output.

Unfortunately, neural network can't provide any uncertainty estimation on its outputs, just a prediction, and PINNs inherit this problem. Several methods were proposed to provide an interval prediction also in NNs, like for instance using a drop-out method, but the state of the art for interval prediction in neural network is using a bayesian approach. Bayesian Neural Networks (BNNs) provide a natural extension of the neural networks framework, where we provide a prior distribution for every parameters in the network $\boldsymbol{\theta}$, and update these distribution with the knowledge of data using Bayes's rule, computing then what is called the posterior distribution of $\boldsymbol{\theta}$. For each input now we can compute a distribution for the output, not only a point evaluation (that in this case could be the mean of the distribution), and this distribution has its own uncertainty in terms of variance and standard deviation. Computing directly a posterior distribution is complicated, since it involves some integral computations; for this reason we can use a Markov Chain Monte Carlo (MCMC) method to directly sample from posterior distribution, without having to compute it analytically. As Neural Network, also Physics-Informed Neural Network can be extended to a Bayesian framework leading to what we call Bayesian Physics-Informed Neural Network (BPINN), that apply the same concepts (using also the physical knowledge, in addiction to data-driven model) to Bayesian Neural Network instead of simply Neural Network.

3.3 Bayesian Physics-Informed Neural Network

In the Bayesian framework, we are interested in study how our prior beliefs will change after collecting real data. Denote with $\boldsymbol{\theta}$ all the variables of our surrogate model (function approximator), that is the BNN itself, we can compute for every input $\mathbf{x}$ the relative approximation of both activation times and velocity (consider the Isotropic equation for simplicity):

$$\mathbf{f}^{\theta}(\mathbf{x}) = \left[NN_T^{\theta}(\mathbf{x}), NN_v^{\theta}(\mathbf{x})\right] = \left[T^{\theta}(\mathbf{x}), v^{\theta}(\mathbf{x})\right]. \tag{9}$$

Bayes theorem describes how our prior beliefs are updated after collecting data. The posterior distribution of our parameters will be proportional to likelihood times prior distribution:

$$\mathcal{P}\left(\boldsymbol{\theta}, \sigma_D, \sigma_R | \mathbf{D}^{\theta}, \mathbf{R}^{\theta}\right) \;\; \alpha \;\; \mathcal{P}\left(\mathbf{D}^{\theta} | \boldsymbol{\theta}, \sigma_D\right) \mathcal{P}\left(\mathbf{R}^{\theta} | \boldsymbol{\theta}, \sigma_R\right) \mathcal{P}(\boldsymbol{\theta}) \mathcal{P}(\sigma_D) \mathcal{P}(\sigma_R). \tag{10}$$

In the following subsections, we will describe briefly both likelihoods and priors.

3.3.1 Likelihood function

In the training phase, we are going to use a small dataset of noisy evaluations of activation times only. Let's write this dataset as a vector $\mathbf{D}$ of n_{exact} exact sparse points, that our BNN will use to partially reconstruct the activation times map. As we have said, these data are noisy, and we can imagine that our dataset, which someone provide to us, comes from the original and exact

dataset $\mathbf{D}^*$ plus a white noise, that we can suppose a normal error $\boldsymbol{\epsilon}$ of mean 0 and standard deviation σ (noise level) fixed and unknown:

$$D(i) = D^*(i) + \epsilon(i), \quad where\ \epsilon(i) \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, N_{exact}$$

Let $\mathbf{D}^{\theta}$ be our approximation through the BNN of the noisy data $\mathbf{D}$, we have a likelihood distribution of:

$$\mathcal{P}(\mathbf{D}^{\boldsymbol{\theta}}|\boldsymbol{\theta}, \sigma_D) \sim \mathcal{N}(\mathbf{D}, \Sigma_D) \tag{11}$$

where $\Sigma_D = \sigma_D^2 \mathbb{I}$, and σ_D can be a fixed value or an hyperparameter.

In addition to the dataset $\mathbf{D}$, we know that the underlying relation between T and v is the Eikonal equation. Let us introduce a vector of n_{coll} collocation points, selected in the spatial domain, where we impose that the BNN must fulfill the PDE constraint. Hence, we denote the residual of the equation as a vector $\mathbf{R} = \mathbf{0}$ of n_{coll} collocation points, where we force the BNN to satisfy the PDE constraint. We can compute our approximation of $\mathbf{R}^{\theta}$ starting from $\mathbf{f}^{\theta}(\mathbf{x})$ as

$$\mathbf{R}^{\boldsymbol{\theta}}(\mathbf{x}) = \|\nabla T^{\boldsymbol{\theta}}(\mathbf{x})\| v^{\theta}(\mathbf{x}) - 1 \quad \forall \mathbf{x} \in \{collocation\ \ points\}_{i=1}^{n_{coll}}$$

We can suppose also in this case a Normal distribution over the real value of $\mathbf{R} = \mathbf{0}$ with an error of σ_R:

$$\mathcal{P}(\mathbf{R}^{\boldsymbol{\theta}}|\boldsymbol{\theta}, \sigma_R) \sim \mathcal{N}(\mathbf{0}, \Sigma_R) \tag{12}$$

where $\Sigma_R = \sigma_R^2 \mathbb{I}$, and also in this case σ_R can be fixed or an hyperparameter.

3.3.2 Priors distributions

Let $\boldsymbol{\theta}$ the vector of all the weights of our NN (that contains all the entries of every matrix W and bias vector for each layers). We can define a prior shape for every entries of the vector θ_j and we have to find its posterior distribution, using PINN and data we have.

$$\mathcal{P}(\theta_j) \sim StudentT(\mu^*, \lambda^*, \nu^*) \tag{13}$$

where μ^*, λ^*, ν^* are simply three constant that we choose.
We have to define also a prior for the two variances σ_D^2 and σ_R^2: both can be considered as Inverse Gamma with fixed parameters.

$$\mathcal{P}(\sigma_D^2) \sim Inv - Gamma(\alpha_1, \beta_1) \tag{14}$$

$$\mathcal{P}(\sigma_R^2) \sim Inv - Gamma(\alpha_2, \beta_2) \tag{15}$$

where $\alpha_1 = \alpha_2 = 2$, while β_1 and β_2 encode our uncertainties on both the data we have collected and the PDE equation behind the problem. In this work for example we assume that the Eikonal equation represents well the behaviour of activation times in cardiac tissue, by choosing a small value as $\beta_2 = 10^{-4}$,

while the uncertainty on the data collected could be bigger, so we consider for instance values that range from $\beta_1 = 10^{-4}$ to $\beta_1 = 10^{-2}$. The latter choice affects the resulting priors of σ_D, and consequently the likelihood on the data similarity (11): using in the prior a bigger β_1 (for instance 10^{-2}) result in a larger σ_D, and our Normal distribution in the likelihood will have a large variance.

3.3.3 Posterior distributions

Let's finally compute the posterior distribution of our parameters ($\boldsymbol{\theta}$, as well as σ_D and σ_R if we choose to consider them as hyperparameters) using the exact noisy dataset $\mathbf{D}$ and the residual on the collocation points $\mathbf{R}$, using the likelihoods and priors defined before. We can compute the posterior distribution for θ, σ_D and σ_R simply following the Bayes Rule:

$$\begin{aligned}\mathcal{P}\left(\boldsymbol{\theta}, \sigma_D, \sigma_R | \mathbf{D}^\theta, \mathbf{R}^\theta\right) \;&\alpha\; \mathcal{P}\left(\mathbf{D}^\theta | \boldsymbol{\theta}, \sigma_D\right) \mathcal{P}\left(\mathbf{R}^\theta | \boldsymbol{\theta}, \sigma_R\right) \mathcal{P}\left(\boldsymbol{\theta}\right) \mathcal{P}\left(\sigma_D\right) \mathcal{P}\left(\sigma_R\right) \\ &\alpha\; \mathcal{N}\left(\mathbf{D}^\theta; \mathbf{D}, \Sigma_D\right) \mathcal{N}\left(\mathbf{R}^\theta; \mathbf{0}, \Sigma_R\right) \mathit{StudentT}(\boldsymbol{\theta}; \mu^*, \lambda^*, \nu^*) \\ &\quad IG\left(\sigma_D^2; \alpha_1, \beta_1\right) IG\left(\sigma_R^2; \alpha_2, \beta_2\right)\end{aligned}$$

and from this posterior distribution we can sample different samples of $\theta, \sigma_D, \sigma_R$ using MCMC methods (or method like SVGD). From these sample we compute statistics like mean and standard deviation, as well use them to compute different samples of $\{\left[T^\theta\left(\mathbf{x}\right), v^\theta\left(\mathbf{x}\right)\right]\}_{\theta=1}^M$ and compute the mean and standard deviation of both T and v, that will tell us how much we are close to the real solutions (with the mean) and how much we can rely on our results (thanks to the standard deviation in every point of the original domain).

4 Algorithms

4.1 MCMC methods

In all the algorithms that follow, we need to compute the log posterior probability that can be easily derive from the posterior distribution written above:

$$\begin{aligned} L(\boldsymbol{\theta}) = log(Posterior) =& log\left(p\left(\boldsymbol{D}^{\theta}|\boldsymbol{\theta},\sigma_D\right)\right) + log\left(p\left(\boldsymbol{R}^{\theta}|\boldsymbol{\theta},\sigma_R\right)\right) + \\ & log\left(p\left(\boldsymbol{\theta}\right)\right) + log\left(p\left(\sigma_D\right)\right) + log\left(p\left(\sigma_R\right)\right) \end{aligned} \tag{16}$$

where:

$$log\left(p\left(\boldsymbol{D}^{\theta}|\boldsymbol{\theta},\sigma_D\right)\right) \alpha \left(-\frac{1}{2{\sigma_D}^2}\sum_{j=1}^{n}(D_j^{\theta}-D_j)^2 + \frac{n}{2}log\left(\frac{1}{{\sigma_D}^2}\right)\right) \tag{17}$$

and

$$log\left(p\left(\boldsymbol{R}^{\theta}|\boldsymbol{\theta},\sigma_R\right)\right) \alpha \left(-\frac{1}{2{\sigma_R}^2}\sum_{j=1}^{n}(R_j^{\theta}-0)^2 + \frac{n}{2}log\left(\frac{1}{{\sigma_R}^2}\right)\right). \tag{18}$$

As we have seen in (7), we can also weight the three components of this log posterior (log likelihood of exact data, log likelihood of PDE constraint and log priors) using 3 different parameters $\alpha_{data}, \alpha_{pde}, \alpha_{prior}$, ending up with a modifes log posterior:

$$\begin{aligned} L(\boldsymbol{\theta}) =& \alpha_{data} log\left(p\left(\boldsymbol{D}^{\theta}|\boldsymbol{\theta},\sigma_D\right)\right) + \alpha_{pde} log\left(p\left(\boldsymbol{R}^{\theta}|\boldsymbol{\theta},\sigma_R\right)\right) + \\ & \alpha_{prior} log\left(p\left(\boldsymbol{\theta},\sigma_D,\sigma_R\right)\right) \end{aligned} \tag{19}$$

The final aim of our methods will be sampling from this distribution and be able to compute all the statistics of the outputs.

4.1.1 HMC: Hamiltonian Monte Carlo

The first method that we've implemented is a classical MCMC method, known as Hamiltonian Monte Carlo (HMC) [2], which follows an Hamiltonian dynamics. Suppose that our posterior distribution is defined as:

$$\mathcal{P}\left(\boldsymbol{\theta}|\mathbf{D},\mathbf{R}\right) \simeq exp(-U(\boldsymbol{\theta})) \tag{20}$$

$$U(\boldsymbol{\theta}) = -\,L(\boldsymbol{\theta}) \tag{21}$$

where $L(\boldsymbol{\theta})$ is the log posterior. HMC method introduce an auxiliary momentum variable $\mathbf{r}$ to mimic the Hamiltonian dynamic, building the following Hamiltonian system:

$$H(\boldsymbol{\theta},\mathbf{r}) = U(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{r}^T\mathbf{M}^{-1}\mathbf{r} \tag{22}$$

where $\mathbf{M}$ is an additional mass matrix that can be set to the identity times a fixed constant. We end up in a joint distribution of $\boldsymbol{\theta}$ and $\mathbf{r}$ of:

$$\pi(\boldsymbol{\theta}, \mathbf{r}) \sim exp(-H(\boldsymbol{\theta}, \mathbf{r})) \tag{23}$$

from which we can sample different samples of theta, that belong to the following dynamical system:

$$\begin{cases} d\boldsymbol{\theta} = \mathbf{M}^{-1}\mathbf{r}dt \\ d\mathbf{r} = -\nabla U(\boldsymbol{\theta})dt. \end{cases} \tag{24}$$

We numerically approximate (24) using a "leap frog" method [7], leading to the following time-step for each iteration t from 1 to a fixed L:

$$\begin{cases} \mathbf{r}_t = \mathbf{r}_t - \dfrac{dt}{2}\nabla U(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta_t} + dt\mathbf{M}^{-1}\mathbf{r}_t \\ \mathbf{r}_{t+1} = \mathbf{r}_t - \dfrac{dt}{2}\nabla U(\boldsymbol{\theta}_{t+1}). \end{cases} \tag{25}$$

The HMC algorithm works as follows [13] (consider $\boldsymbol{\theta} = (\boldsymbol{\theta}, \sigma_D, \sigma_R)$ for simplicity, if they are trainable too, and choose $\mathbf{M} = \mathbb{I}$ as the identity):

- Initialize $\boldsymbol{\theta}^{t_0}$, fix the number of iterations N, the number of samples M to collect (after burn-in), the number of leap-frog steps L and the step size dt
- for every k in $1, \ldots, N$:
  1. Sample $\mathbf{r}^{t_{k-1}} \sim \mathcal{N}(\mathbf{0}, \mathbb{I})$
  2. $(\boldsymbol{\theta}_0, \mathbf{r}_0) = (\boldsymbol{\theta}^{t_{k-1}}, \mathbf{r}^{t_{k-1}})$
  3. for i in $0, \ldots, (L-1)$:

$$\begin{aligned} \mathbf{r}_i &= \mathbf{r}_i - \frac{dt}{2}\nabla U(\boldsymbol{\theta}_i) \\ \boldsymbol{\theta}_{i+1} &= \boldsymbol{\theta_i} + dt\mathbf{r}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - \frac{dt}{2}\nabla U(\boldsymbol{\theta}_{i+1}) \end{aligned}$$

  4. sample $p \sim Uniform(0, 1)$
  5. compute $\alpha = \min(1,\ \exp(H(\boldsymbol{\theta}_L, \mathbf{r}_L) - H(\boldsymbol{\theta}^{t_{k-1}}, \mathbf{r}^{t_{k-1}})))$
  6. If $p \geq \alpha$, then $\boldsymbol{\theta}^{t_k} = \boldsymbol{\theta_L}$,
     else $\boldsymbol{\theta}^{t_k} = \boldsymbol{\theta}^{t_{k-1}}$

and finally compute all the statistics we need using $\{\boldsymbol{\theta}^{t_i}\}_{i=N-M+1}^{N}$.

4.2 SVGD: Stein Variational Gradient Descent

Since the computation of the posterior with standard MCMC methods can become infeasible, we also employ the Stein Variational Gradient Descent algorithm (SVGD) [4] with a finite number of neural networks (called "particles"), for example N=30, that will approximate the posterior distribution (of course the larger N, the better the approximation, however, implying a growing computational time).

The SVGD algorithm works as follows, as shown in [12]:

- Initialize N neural networks weights $\boldsymbol{\theta^i}$ $\forall i = 1, \dots, N$, N σ_D^i and σ_R^i
- For every epochs $e = 1, \dots, E$:
  1. Compute the log posterior $L(\boldsymbol{\theta}^i)$ $\forall i = 1 \dots, N$
  2. Compute the gradients $\nabla_{\boldsymbol{\theta}^i} L(\boldsymbol{\theta}^i)$ by back-propagation
  3. Compute

$$\phi(\boldsymbol{\theta}^i) = \frac{1}{N} \sum_{j=1}^{N} \left[k(\boldsymbol{\theta}^i, \boldsymbol{\theta}^j) \nabla_{\boldsymbol{\theta}^i} L(\boldsymbol{\theta}^i) + \nabla_{\boldsymbol{\theta}^i} k(\boldsymbol{\theta}^i, \boldsymbol{\theta}^j) \right] \quad \forall i = 1 \dots, N \tag{26}$$

  4. Update $\boldsymbol{\theta}^i = \boldsymbol{\theta}^i + \epsilon \phi(\boldsymbol{\theta}^i)$ or use a Stochastic Gradient Descent method, $\forall i = 1, \dots, N$
  5. Update σ_D^i and σ_R^i (simply through back-propagation or use also the SVGD methods), $\forall i = 1, \dots, N$

where $k(x_i, x_j)$ is the Radial Basis Function (RBF) kernel [6], namely:

$$k(x_i, x_j) = exp(-\frac{1}{h} \|x_i - x_j\|^2)$$

for a constant bandwidth $h > 0$.
The key passage in this method is (26), where we modify the gradients of $\boldsymbol{\theta}$ from every N neural networks to approximate the posterior distribution; the first term drives the particles towards the high probability areas of $L(\theta)$ by following a smoothed gradient direction, which is the weighted sum of the gradients of all the particles weighted by the kernel function $k(x_i, x_j)$, while the second terms (gradients of k) acts as a repulsive force that prevents all particles to collapse to a single point, namely the MAP of the posterior distribution.

5 Libraries

The code is written in Python, version 3.8.5, but it should work with every version of Python from 3.5 to 3.8 (while if you are using Python 2 you could experience some compatibility problems). You can download the correct python version from `https://www.python.org/downloads/`.
The code relies on different external libraries, you can find a list with all the correct versions in "requirement.txt", that are:

- Tensorflow
- Tensorflow_probability
- Numpy
- Matplotlib
- (optional, 3D plot) Mayavi.

Tensorflow is by far the most important and exploited library in the code. It provides all the neural network architectures using Keras, that is integrated in Tensorflow since the version 2 of the library; the Adam optimizer, the dataloader (also for minibatch training) and the tf.tensor module. For this project we have used tensorflow version 2.5.0; it can be installed following the instruction in `https://www.tensorflow.org/install` (to install it using pip, it is required a version $\geq$ 19.0). We have chosen to use tensorflow version 2, despite the majority of the codes of PINN are written in tensorflow version 1, because it is newer and has already inside all the Keras library, and is available the eager execution, instead of the graph execution that was mandatory in TF v1, that is more intuitive and simple, and easier to debug (even if it is usually slower, but we can simply switch to graph execution by adding the decorator "@tf.function" before a function).
If a CUDA-compatible GPU is available and all the CUDA packages are installed (find here `https://www.tensorflow.org/install/gpu` all the detail), tensorflow will automatically download the gpu-accelerated version, or tensorflow-cpu on the contrary. For the purpose of this code, the gpu-acceleration doesn't make a big difference, since the neural network model are usually too small to see a big advantage (see the relative Appendix for more details).

Tensorflow_probability is a library built on Tensorflow that makes easy to use statistical and probabilistic models inside the Tensorflow framework. We use it to sample from particular distributions like the Gamma distribution for instance. All the instruction to install it can be found in `https://www.tensorflow.org/probability/install`; in the code we use it in the version 0.11.1 .

Another key library adopted in this project is Numpy, one of the most used python library for Scientific computing, useful for the handling of n-dimensional arrays and matrix, also for the numpy.random module. It is available here

`https://numpy.org/install/`, in the code we use it in version 1.19.4 .

Finally, to plot all the results (in 1D but also 2D and 3D) we rely on the Matplotlib library, that can be found here `https://matplotlib.org/users/installing.html` in version 3.2.0. For interactive plot during execution (and not only plot saved in images), other dependencies are required beside the required ones for Matplotlib, such as "Tk", "PyQt4" etc. The complete list is in the previous url.

Optionally, you can install also the library "Mayavi" for 3D plot using the results of the algorithm (`https://docs.enthought.com/mayavi/mayavi/`, we are using the version 4.7.2).

All the mandatory requirements are listed in "requirements.txt", which can be installed using pip (Package installation manager for Python, `https://pip.pypa.io/en/stable/`):

```
$ pip install -r requirements.txt
or
$ pip3 install -r requirements.txt
```

6 Code Organization

The main code is organized as follow: in the main folder ("BPINN-Inverse-Eikonal-UQ") you can find the main file "mainsolve.py", a README.md file that briefly illustrates the project and a "requirements.txt" with a list of all the dependencies needed, plus some subfolders:

- "utils" that contains all the classes and functions used in the project
- "data" that contains all the different datasets for our experiments
- "config" that contains all the file json that we are going to use to read the parameters for each experiment
- result folders, namely:
  - 1D-isotropic-Eikonal
  - 2D-isotropic-Eikonal
  - 3D-isotropic-Eikonal
  - 2D-anisotropic-Eikonal

  that will contain all our experiments, subdivided with respect to input dimension and type of equation (isotropic or anisotropic).

To run the code, it is necessary to specify at least the method you want to use. For instance, this could be the simpler way to run the code from command line (after you change the directory to the main directory "BPINN-Inverse-Eikonal-UQ"):

```
$ python mainsolve.py --method METHOD
```

where as method flag METHOD you can choose between HMC or SVGD.
Another additional flag is "config", that is used to specify the json file where we have specified all the parameters. This file have to be placed in the "config" subfolder and have a particular form. To run the code with a particular configuration:

```
$ python mainsolve.py --method METHOD --config yourconfig.json
```

The config flag is not mandatory since if nothing is provided, the code will use the "default.json" file, placed in "config" folder.
In addition to these two flags, it is possible to specify all the parameters (that can be found in every json config file) directly from command line, and the code will select those values, discarding the values presented in the json file; this can be useful when running different experiments with all the same set of parameters but changing only one, for instance the noise level in exact data.
In this way it is not needed to change and made copies of your original json file, but it is possible to just add the flag [–noise_lv NOISE_LV] directly in the command line. More about that in the "Parameters" subsection.

6.1 Overview of the code

The script **"mainsolve.py"** is the main of the code and call all the other classes in the "utils" folder. It contains the workflow of the project.
First, we manage the parameters in input using both the json file and the arg parser in **"args.py"** (using the python module argparse) and then create an object of custom class param, defined in **"param.py"**, that stores all the parameters we are going to use in the following passages.
After creating all the directories to store the results, using the method create_directories() that can be found in **"helpers.py"**, we build an object of "dataset_class" that can be found in **"dataset_creation.py"**, that has the task to build every datasets we need (domain, collocation, exact and noisy datasets) if we provide the analytical function to compute the activation times and conduction velocities, or load them directly if we provide them (everything can be found in the "data" subfolder).
Then we construct the pde_constrain (using classes in **"pde_constrain.py"**) we are going to use in the following bayes_nn class, we can choose between isotropic or anisotropic in accordance to which PDE we want to solve.
Then, we construct the bayesian neural network, that can be found in **"BayesNN.py"**, where there is a father class BayesNN and two child classes, through python inheritance, called "MCMC_BayesNN" and "SVGD_BayesNN". The distinction between MCMC methods and SVGD is needed because in SVGD we are going to approximate the posterior distribution using a number N of different neural networks to be trained at the same time during each epoch, so it has some changes in every methods. The neural network is built using the class Net in **"FCN.py"**, where we simply use the tf.keras.Sequential module and we add some useful methods for updating the weights.
After building the right Bayesian neural network, we need to instantiate the algorithm for training, but before we need a data loader for mini-batch training of collocation points, that are usually much more respect to the exact noisy data: for this, we create an object of dataloader class (that is defined in **"dataloader.py"**).
Finally, we build our selected method to train the bayesian physics-informed neural network: we have two different methods, with relatively classes defined in **"HMC_MCMC.py"** and **"SVGD.py"**, and using the method "train_all()".
After the training, all the results (errors, uncertainty quantifiation and plots) are computed using **"compute_error.py"** and **"plotter.py"**.

6.2 Parameters

The parameters for the code are organized in a nested python dictionary written in a json file, that must be placed in "config" folder, and have the following keys:

- "architecture": features of our neural network architecture (number of hidden layers and number of neurons for each hidden layer);
- "experiment": a dictionary with all the features to create our datasets

(which experiment we are going to use, which proportion of exact and collocation data we want, the noise we want to add to the exact data and finally the batch size for collocation points);

- "param": weights for the three log posterior parts. It includes the eikonal loglikelihood, data loglikelihood (also called log joint) and prior log distribution, a parameter for penalizing high gradients in conduction velocity (param2loss) and a random seed for numpy random generator;
- "sigmas": parameters for σ_D and σ_R, if we want to consider them as hyperparameters;
- "utils": parameters for verbosity and saving results;
- two different keys, one for each methods, "HMC" and "SVGD", where we have a dictionary with all the parameters needed by both methods, that are:
  - "HMC": "N_HMC", "M_HMC", "L_HMC", "dt_HMC";
  - "SVGD": "n_samples", "epochs", "lr", "lr_noise", "param_repulsivity".

The code requires to specify the chosen method directly from command line using the flag –method (and then HMC or SVGD), it is possible to chose the json file where all these parameters are setted using the flag [–config yourjsonfile.json] (it is not mandatory, if not provided the code will use the default.json file included in the "config" folder); in addiction to this, we have implemented also an additional way to provide the parameters to the code directly from the command line: in **args.py** we have provided an arg parser that collect, in addition to method and config arguments, also every parameters you can find in a json file. This might be not so relevant, since the parameters are already specified in your config file, however it proves very useful in order to build a config file, when only some parameters must be varies, like, e.g., the proportion of exact data or noise level. In this way, it is not required to create a new json file, but it is possible to "overspecify" parameters directly from the command-line, and the code will use that value above the value described in json file.
The parameters read by the config json file and additional parameters collected with the arg parser are then passed to a new class called param (defined in **param.py**), that returns a class with an attribute for each key of the outer dictionary in json file. With the method "_update()" we update the parameters using commad-line additional parameters, and then we add some other useful characteristics using the three dictionaries defined below, namely "n_input", "pde" and "dataset_type". These three dictionaries, given as key the name of dataset we are going to use, return the input dimension, which type of pde we want to solve (isotropic or anisotropic) and the dataset type (if we compute the dataset using the exact analytical functions or if we have collected the data from an Eikonal forward solver, for instance using the library PyKonal, described in the Appendix).

6.3 Data, Dataset_class and dataloader

All the different experiment data are stored in the subfolder "data", organized in different folders each containing numpy vectors for each input and output: for instance, in "prolate3D" there are "x.npy", "y.npy", "z.npy", "at.npy" and "v.npy", since this experiment comes from a forward numerical problem (using PyKonal). On the other hand, if the experiment is "analytical", we have, instead of the numpy vectors, a python script named "analytical_functions.py" where we have defined the two functions, one to compute the activation times, one to compute conduction velocities, given inputs. The datasets are managed in the code thanks to the class "dataset_class", defined in **"dataset_class.py"**. We are going to create three different datasets, that we are going to use in the project:

- domain dataset, that is the whole domain (dimension of n_{dom}) and we are going to use it to compute errors and plot the results;
- collocation dataset, which has a dimension of $n_{coll} = prop_{coll} n_{dom}$ (but usually we use to whole domain dataset, so with $prop_{coll} = 1$), and will be our dataset where we impose the pde constraint;
- exact dataset (and then we compute Exact noisy dataset), which has a dimension of $n_{ex} = prop_{ex} n_{dom}$ (and usually $n_{ex} << n_{coll}$), that will be our exact dataset where we impose the data-driven constraint.

For every dataset, we create three different attributes: inputs, T and CV. The method for building all the three datasets acts differently if the experiment comes from a numerical solver or if we provide the analytical solution:

- in the case where we have a numerical experiment data, we load the numpy vectors and store them in the domain dataset attributes (inputs_dom, T_dom and CV_dom, and n_{dom} is the length of one of them); then the exact dataset and collocation dataset are created employing a random sampling from the domain dataset, using the right numerosity, thanks to the method "numpy.random.randint()";
- if we provide the analytical functions, we load both the functions in our dataset_class, then we create all the three datasets input using numpy.meshgrid() (for domain dataset input) or numpy.random.random() (for exact and collocation dataset inputs), each with the correct cardinality and after that we compute the exact activation times and conduction velocity using the functions we have loaded (in this case we need to specify at least n_{dom}, and this is done, only for analytical case, in param class).

All these passages are managed by the class method "build_dataset()" only the first time we need the datasets (using an attribute flag named _flag_dataset_build that becomes True after the first time). The method "build_noisy_dataset" creates the dataset of noisy exact starting from the exact dataset and adding a noise of level "self.noise_lv" just the first time we call it (using a flag in the

same way as before).
Since we want to employ a mini-batch training (at least for the collocation points, that are usually much more than the exact points), we need a dataloader: we build a custom class named dataloader, that is defined in **"dataloader.py"** and take advantage of all the methods for dataloading available in tensorflow library. In the class method "dataload_collocation()" we first collect the input (we only it) of collocation dataset from an instance of dataset_class using the getter method "get_coll_data()", then we use the method tensorflow.data.Dataset.from_tensor_slices of tensorflow, to convert it from numpy n-array to tensorflow dataset. Thanks to shuffle() and batch() tensorflow methods, we build a dataloader that gives us the collocation points in batch of size self.batch_size and it reshuffle the dataset after each iteration (after it returns all the batches).

6.4 BayesNN and FCN

In **FCN.py** we define our custom class to build a Fully Connected Neural Network "Net"; we rely on the module tf.keras.Sequential(), that builds a neural network adding one after one all the needed layers, and we place the network in the attribute "self.features". The architecture of our FCN is the following:

- n_input: input dimension (depends on the experiment we choose, can be 1D, 2D or 3D);
- n_hidden_layers: number of hidden layers in the network;
- n_neurons: number of neurons each hidden layers has;
- n_output: output dimension, that is equal to 1+n_output_vel (1 for activation times, and n_output_vel is equal to 1 in isotropic case, or > 1 in anisotropic case).

In Figure 2, a rappresentation of a FCN for 3D isotropic Eikonal (n_input=3, n_output_vel=2) and with n_hidden_layers=3, n_neurons=5 is reported.

Let's analyze briefly how the parameters of the network are organized, since we are going to use them directly in both the algorithms (HMC and SVGD): we are relying on the tensorflow Sequential attribute "trainable_weights", that is a list of matrix and vector for each layer in the network. For instance, in the network showed in 2, where we have 3 hidden layers (5 layers if we count also input and output, L=5), our $\boldsymbol{\theta}$ (that collects all the trainable parameters of the NN) will be a list of 8 items (list length $= 2(L-1) = 2(N_{hidden\_layer}+1)$):

$$[\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{W}_3, \mathbf{b}_3, \mathbf{W}_4, \mathbf{b}_4] \tag{27}$$

where $\mathbf{W}_l$ is the weight matrix of shape $= (n_{l-1}, n_l)$ and $\mathbf{b_l}$ is the bias vector of dimension $= n_l$, $\forall l = 1, \ldots, L-1$.
In addition to some other simply useful methods, like the one for counting the number of total parameters and one that returns the architecture of the network,

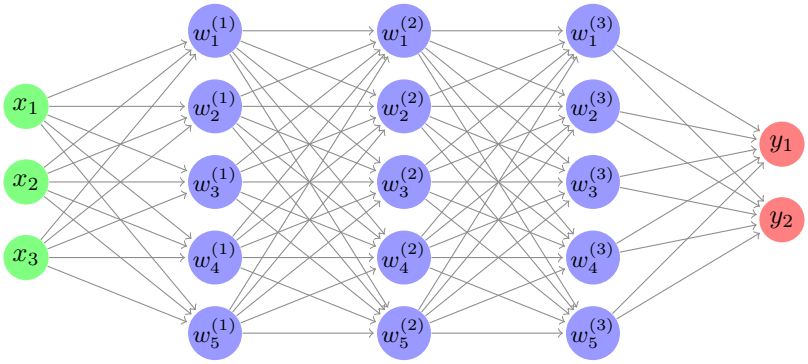


Figure 2: FCN Architecture for 3D isotropic Eikonal

we add a method for updating all the weights in the neural network (for the HMC algorithm), giving in input a list of tensors and vectors for each layer. The method, called "update_weights", exploits the method of the tensorflow class tf.keras.layers.Layer set_weights() for each layer in self.features.layers, that require in input a tuple of Weight matrix and bias vector (W,b). As shown below, we loop over the list param using an index i=0 and update it every step by adding 2.

```
def update_weights(self, param):
    """!
    Update the weights giving a list of tensors
    @param param list of tf tensor represent the
    new weights for the network
    """
    i = 0
    for layer in self.features.layers:
        # use the method set_weights for each layer
        # first param is the matrix W,
        # second the bias vector b
        layer.set_weights((param[i], param[i+1]))
        i+=2
```

The BayesNN class is probably the most important class of the implemented code. It collects in its attributes a lot of different classes we have created, like FCN but also "pde_constraint" and "other_trainable_param" that we'll briefly explain after; most importantly, here we define all the methods to compute the log posterior distribution, including forward pass of the inputs and relatively derivative using the Automatic Differentiation (thanks to tensorflow.GradientTape) to compute ∇T (more details about Automatic Differentiation in the relatively appendix). The class BayesNN is an abstract class to which we create two child classes, MCMC_BayesNN and SVGD_BayesNN, as shown in 3. This allows us to distinguish between all the classical MCMC methods and

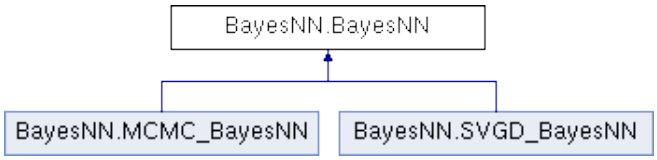


Figure 3: BayesNN classes

the SVGD method need to be present also in the definition of Bayesian Neural Network class (and not only in the training algorithm of course) because the SVGD methods, differently from every other MCMC methods, need to store a finite number of Neural Networks to be train at the same time in every epochs; for this reason we have to override some methods.
Among all its attributes (for instance we store all the prior parameters for both $\boldsymbol{\theta}$ and σ_D, σ_R), "self.nnets" is a list of instances of Net class (defined in FCN.py). In father class BayesNN we add the first instance of Net to the list:

```
self.nnets.append(Net(self.n_input, architecture["n_layers"],
            architecture["n_neurons"], self.n_output_vel+1)).
```

For the MCMC methods we need just a list of a single element, while in SVGD_BayesNN we have to add all the others (num. of neural networks - 1) Nets. So, only in SVGD_BayesNN, we have added also these lines:

```
for i in  range(self.num_neural_networks-1):
    new_instance = Net(nFeature, architecture["n_layers"],
            architecture["n_neurons"], self.n_output_vel+1)
    self.nnets.append(new_instance) # append the i-th NN
```

Another two very important attributes of BayesNN class are:

- self.log_betas : instance of class trainable_param that stores variables for log_betas (where $\beta_D = \frac{1}{\sigma_D^2}$ and $\beta_R = \frac{1}{\sigma_R^2}$)
- self.pde_constraint : instance of pde_constraint class that will provide the computation of PDE constraint.

6.4.1 Posterior

As shown in (16), we can decompose the log posterior in three components:

$$log\left(p\left(\boldsymbol{D}^{\theta}|\boldsymbol{\theta},\sigma_D\right)\right) + log\left(p\left(\boldsymbol{R}^{\theta}|\boldsymbol{\theta},\sigma_R\right)\right) + log\left(p\left(\boldsymbol{\theta},\sigma_D,\sigma_R\right)\right). \tag{28}$$

In BayesNN we provide two methods to compute all the three components, log_joint() that computes log likelihood of exact noisy data and log prior, and pde_log_loss() that compute the log likelihood of the PDE constraint. To compute the log likelihood of exact data (we can refer to equation (17)), we create a list and append the evaluation of the following computation (as before for the attribute self.nnets, we are going to have a list of just one element for every

MCMC methods, and a list of N=num. neural network evaluations for SVGD, one for each neural network stored in self.nnets):

```
log_likelihood = []
for i in  range(self.num_neural_networks):
    log_likelihood.append(
    -0.50*tf.math.exp(self.log_betas.log_betaD[i])*
    tf.reduce_sum((target[:,0] - output[:,i])**2)
    + 0.50 * (tf.size(target[:,0], out_type = tf.dtypes.float64))
    * self.log_betas.log_betaD[i]).
```

Here output[:,i] is prediction of the sparse input using the i-th NN stored in self.nnets (that is, as said before, $i = 0$ for MCMC, $i = 0, \dots, N-1$ for SVGD) and target[:,0] are the exact noisy activation times; then, we use the equation (17) with the difference that, instead of knowing σ_D, we have just $\log(\beta_D)$, where $\beta_D = \frac{1}{\sigma_D^2}$.
Following the same exact rule, we can compute log likelihood of PDE constraint, where instead having the square difference of target and output, we have the difference between the residual of the PDE computing with our Neural network (using pde_constrain class method "compute_error") and a vector of zeros, that represent the exact evaluation of the residual (and of course instead of log_betaD we use log_betaR).
Finally, to compute the log prior of our parameters (that are all the neural networks weights, plus log_betaD and log_betaR if we consider them as hyper-parameters), we just apply logarithm to the prior distributions defined in (13), (14) and (15).

6.4.2 Gradients

Another fundamental method in this class is "_gradients()", that is responsible of the computation of derivatives (of T and v) with respect to the inputs (x, y and z in 3D case for instance). To obtain that, we don't use any kind of numerical differentiation, but we rely on the Automatic Differentiation method, already employed in tensorflow thanks to tf.GradientTape class. Automatic Differentiation is a method to compute exact derivative of output with respect to input employing the chain rule of derivation and knowing only the derivative of basic arithmetic operations and functions. Further details on the way Automatic Differentiation works are reported in the Appendix.
The class methods return two list of gradients evaluations (a list of vectors for each input dimension), one for T and one for v; in SVGD_BayesNN we have to override the methods since we call a different forward pass, namely "_forward_stacked()" to evaluate the derivative for every neural networks in self.nnets.
This methods, like basically every methods that use tf.GradientTape in this project, has the python decorator "@tf.functions" before its definition: we are telling tensorflow to use its "Graph computation" instead of the new tensorflow 2 "eager computation" mode. This will decrease drastically the computational

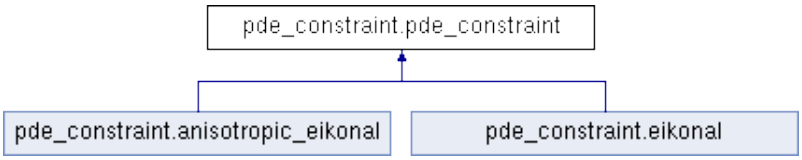


Figure 4: Pde constraint classes

time, since graph execution is way faster, typically when we are computing gradients (not only with respect to the inputs, but also with respect to the parameters, as we will see in the next sections). More details can be found in Appendix C.

6.5 pde_constraint and other_trainable_param

The parent class pde_constraint implement the concept of a PDE constraint. It is an abstract class, since the principal method, that is "compute_pde_losses()" has to be override by all the children classes, that employ a specific pde. In this project we have built only two child classes, that are "eikonal" (isotropic) and "anisotropic_eikonal", that are build by inheritance from "pde_constraint" as shown in figure 4.

Even if it seems not so useful to build a parent class and then child classes for this purpose, we have done this in order to increase the possibility to extend the project to other Eikonal pde types, like the Eikonal-Diffusion equation [3], or even better to other pde problem (in this case some minor changes are also required).

The "trainable_param" class is useful to store log_betaD and log_betaR, in particular is useful if we set them to "trainable" (in other terms, if we consider them as hyperparameters) since we have build some methods to update them easily. In MCMC case, we use 1 value for each, while in SVGD we have N values for each one. We rely on tensorflow-probability library to sample them from the right distributions, that is a Gamma for beta, and then take the logarithm to compute log_beta (since σ_D^2 and σ_R^2 comes from Inverse Gamma, as said in (14) and (15)).

6.6 Algorithms: HMC

The class HMC_MCMC, defined in "**HMC_MCMC.py**", implements the Hamiltonian Monte Carlo algorithm for training a Bayesian Neural Network. We provide to the class a MCMC_BayesNN instance during costruction (passed by reference, as standard in Python), that will be stored in its attribute "self.bayes_nn" and trained using HMC.
The key method, a "public" method called from main, is "train_all()", that starts the training (with in input a parameter for verbosity; all the other parameters for the training, like N, M, dt and so on, were already passed during

construction). The method works as described in section 4.1.1; after an initialization of $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_0$ (the first using the initial weights in our network, the second just by a copy), for every iteration in $1, \dots, N$, after computing $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_0$ as before, we compute $\mathbf{r}$ by sampling from a $\mathcal{N}(0, 1)$ and then make a copy $\mathbf{r}_0$.
The vector $\mathbf{r}$ (called "rr" in the code) is a list of tensors and vectors, just as $\boldsymbol{\theta}$: for every layer $l = 1, \dots, n_{totalLayers} - 1$ in neural network we append a matrix (of the shape of $\mathbf{W}_l$) and then a vector (of shape of $\mathbf{b}_l$) of samples from a Normal distribution.
Then we loop over every step in $1, \dots, L$, where L here represent the leapfrog steps, and then we loop over the batches of collocation points (if we want to employ a mini-batch training) and perform a complete Leapfrog step, as described in (25).
First, we compute $\nabla U(\boldsymbol{\theta})$ using the class method "grad_U_theta", that basically compute the log posterior using the relative BayesNN methods, then compute the derivative of $U(\boldsymbol{\theta}) = -\log(posterior(\boldsymbol{\theta}))$ using Automatic Differentiation (with a tf.GradientTape).
After this step, we update $\mathbf{r}$ using the simple function "list_update()" by a step of $-\frac{dt}{2}\nabla U(\boldsymbol{\theta})$.
Then we update $\boldsymbol{\theta}$ using list_update() with a step of $dt\mathbf{r}$, and of course we need to update also the weights in our NN: for this reason, we call the FCN method "update_weights()" with the new $\boldsymbol{\theta}$ we have already computed.
The last step of our Leapfrog is similar to the first one, we compute $\nabla U(\boldsymbol{\theta})$ and use it to update $\mathbf{r}$.

```
## for every step in 1,...,L (L = LEAPFROG STEPS)
for s in range(self.L):
    ## iterate over all the batches of collocation data
    for batch_idx,inputs in enumerate(self.train_loader):

        ########################################################
        ##############      Leapfrog step    ###################
        ########################################################

        ## backpropagation using method grad_U_theta
        grad_theta,u_theta,log_likelihood,log_prior_w,log_eq, \
        losses = self.grad_U_theta(sp_inputs, sp_target, inputs)

        ## update rr = rr - (dt/2)*grad_theta
        rr = list_update(rr, grad_theta, -self.dt/2)

        ## update theta = theta + dt*rr
        theta = list_update(theta, rr, self.dt)
        ## update the weights in the NN with the new theta
        self.bayes_nn.nnets[0].update_weights(theta)

        ## backpropagation using method grad_U_theta
        ## (now with the new theta)
        grad_theta,u_theta,log_likelihood,log_prior_w,log_eq, \
```

```
        losses = self.grad_U_theta(sp_inputs, sp_target, inputs)

        ## update rr = rr - (dt/2)*grad_theta
        rr = list_update(rr, grad_theta, -self.dt/2)
```

After all the steps in the two nested for loops, we end up with a new couple $\boldsymbol{\theta}$ and $\mathbf{r}$, that now are different from the previous $\boldsymbol{\theta}_0$ and $\mathbf{r}_0$: we need here the "accepted/rejected" step of our MCMC method.
After computing $\alpha \in [0, 1]$ through class method "alpha_fun()", we sample $p \sim Unif(0, 1)$ and we accept the new values of $\boldsymbol{\theta}$ and $\mathbf{r}$ only if $p > \alpha$, otherwise we keep the previous values. Finally, we store all the thetas in a list in an attribute of MCMC_BayesNN from which we will compute mean and standard deviations of output. If we consider also log_betaD and log_betaR as trainable, we have to repeat all the passages for theta also for them, but with a different dt (called dt_noise).

6.7 Algorithms: SVGD

As the previous class for HMC, SVGD class implement the training algorithm for Stein Variational Gradient Descent, described in 4.2. The public method, that will be called from mainsolve.py, is called exactly the same as HMC, "train_all()", with the same input (verbosity). As before, we have two nested for loops, the outer for epoch in number of epochs, the second loop over all the batches of collocation points (if we want to implement a mini-batch training). Inside the nested for loops, we perform a complete step of SVGD: first, we compute $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ (where $L(\boldsymbol{\theta})$ is the log posterior, (16)) by back propagation using class method "compute_backprop_gradients()", that use Automatic Differentiation to compute gradients wrt $\boldsymbol{\theta}$ (tf.GradientTape).
Then we transform both $\boldsymbol{\theta}$ and $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ from list of tensor and vector, as explained in (27), to a flat tensor of shape = (number of neural networks, total number of parameters in $\boldsymbol{\theta}$). Total number of parameters in $\boldsymbol{\theta}$ is simply the count of all the entries of every item in the list (27), so for instance in the architecture showed in 2, we have a total of 92 parameters.
We need this transformation because we have to compute the RBF Kernel and its derivative, to update properly the gradients, as shown in (26). To compute Kxx and DxKxx we rely on the class method "_Kxx_dxKxx()".
After this passage, we reconstruct the original form of $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ and the use the Adam optimizer (with a learning rate of "self.lr") to update $\boldsymbol{\theta}$ for each neural networks in self.bayesNN.nnets.

```
# loop over every epochs
for epoch in range(self.epochs):
  # loop over self.train_loader: minibatch training
  for batch_idx,inputs in enumerate(self.train_loader):

    # use the method self.compute_backprop_gradients()
    # to compute the gradients of theta (and losses)
    grad_parameters, ... = \
```

```
self.compute_backprop_gradients(sp_inputs, sp_target, inputs)

# reshape theta and grad_theta from lists to tensors,
# in order to compute Kxx and dxKxx

## build a flat vector of theta
theta = self.bayes_nn.get_trainable_weights_flatten()
## build a flat vector of grad_theta
grad_theta = get_trainable_weights_flatten(grad_parameters)

######## SVGD modify the gradients of theta #########
# calculating the kernel matrix and its gradients
Kxx, dxKxx = self._Kxx_dxKxx(theta)
grad_logp = tf.linalg.matmul(Kxx, grad_theta)
### the minus sign is needed for apply_gradients ###
grad_theta = -(grad_logp + dxKxx)/self.num_neural_networks

# reshape back grad_theta from
# shape=(num_neural_networks, num_total_parameters_theta)
# to (num_neural_networks, *list* )
grad_theta_list=self.from_vector_to_parameter(grad_theta.numpy())
theta = self.bayes_nn.get_trainable_weights()

# optimizer step: apply_gradients of grad_theta_list to theta
for i in  range(self.num_neural_networks):
    # optimizer[0] is the Adam optimizer
    # for theta (learning_rate = self.lr)
    self.optimizers[0].apply_gradients(
     zip(grad_theta_list[i], theta[i]))
```

As for HMC, if we consider also log_betaD and log_betaR as trainable, we have to repeat all the passages for theta also for them, but with a different learning rate (called lr_noise).

6.8 Compute_error and plotter

The class compute_error compute errors and Uncertainty Quantification of our predictions, while in plotter we collect all the functions that we are going to use to plot our result.
Errors and UQ are computed using all the domain dataset for both activation times and conduction velocities: we use as metric a $\mathcal{L}^2$ relative error

$$\begin{aligned} L^2_{rel\ err}(T^{\boldsymbol{\theta}}) &= \frac{\sum_{i=1}^{n_{dom}}(T_i^{\boldsymbol{\theta}} - T_i^{exact})^2}{\sum_{i=1}^{n_{dom}}(T_i^{exact})^2} \\ L^2_{rel\ err}(v^{\boldsymbol{\theta}}) &= \frac{\sum_{i=1}^{n_{dom}}(v_i^{\boldsymbol{\theta}} - v_i^{exact})^2}{\sum_{i=1}^{n_{dom}}(v_i^{exact})^2} \end{aligned} \tag{29}$$

while for Uncertainty Quantification we first compute (numerically) the standard deviation on every input $\mathbf{x}_i \; \forall i = 1, \dots, n_{dom}$ of our predictions:

$$\begin{aligned} std(T_i^{\boldsymbol{\theta}}) =& \sqrt{\frac{1}{M} \sum_{m=1}^{M} (T_i^m - mean(T_i))^2} \quad \forall i = 1, \dots, n_{dom} \\ std(v_i^{\boldsymbol{\theta}}) =& \sqrt{\frac{1}{M} \sum_{m=1}^{M} (v_i^m - mean(v_i))^2} \quad \forall i = 1, \dots, n_{dom} \end{aligned} \tag{30}$$

where $m = 1, \dots, M$ are our sample predictions on that particular $\mathbf{x}_i$, and $mean(T_i) = \frac{1}{M} \sum_{m=1}^{M} (T_i^m)$ and $mean(v_i) = \frac{1}{M} \sum_{m=1}^{M} (v_i^m)$ are the prediction means.

We then compute the mean standard deviation in all the domain and the maximum standard deviation in all the domain:

$$\begin{aligned} ST^{mean}(T^{\boldsymbol{\theta}}) =& \frac{1}{n_{dom}} \sum_{i=1}^{n_{dom}} std(T_i^{\boldsymbol{\theta}}) \\ ST^{max}(T^{\boldsymbol{\theta}}) =& \max_{i=1,\dots,n_{dom}} std(T_i^{\boldsymbol{\theta}}) \\ ST^{mean}(v^{\boldsymbol{\theta}}) =& \frac{1}{n_{dom}} \sum_{i=1}^{n_{dom}} std(v_i^{\boldsymbol{\theta}}) \\ ST^{max}(v^{\boldsymbol{\theta}}) =& \max_{i=1,\dots,n_{dom}} std(v_i^{\boldsymbol{\theta}}). \end{aligned} \tag{31}$$

In plotter, all the plots of course are different by the input dimension (1D, 2D or 3D), but also if we are using an analytical dataset or a real dataset, if we are using HMC or SVGD (in particular in plot_all() functions) and finally if we are using an Isotropic equation or an Anisotropic ones.

7 Numerical Experiments

All the following experiments are performed with HMC. A numerical comparison with SVGD is described in section 7.6.

7.1 1D Uncertainty Quantification

The first experiment is a 1D Eikonal equation in the (0,1) domain, with an exponential behaviour of the activation times and a source located at $x = 0$. In this case the analytical solution is:

$$\begin{aligned} T(x) =& 1 - e^{-2x} \\ v(x) =& \frac{1}{2} e^{2x}. \end{aligned} \tag{32}$$

We can easily prove that with these choices for the activation time T(x) and the conduction velocity v(x) we satisfy the Eikonal equation, that in this case is simply:

$$\left| \frac{\partial T(x)}{\partial x} \right| = \frac{1}{v(x)}. \tag{33}$$

We analyze this case with the same parameters (that can be found in the json file 1D_HMC.json in the project folder), varying the number of data and the noise level. We consider:

- architecture: 5 hidden layers, 100 neurons each;
- dataset: 1000 collocation points, 10/20/40 exact data, noise level 0.01/0.05/0.10, no mini-batch training;
- parameters: $\alpha_{pde} = 1$, $\alpha_{data} = 20$, $\alpha_{prior} = 0.5$;
- σ_D and σ_R not trainable;
- HMC with N=2000, M=1000, L=10 and dt=10^-3.

It is possible to run this first test case with the following command:

```
$ python mainsolve.py --method HMC --config 1D_HMC.json
--noise_lv NOISE_LV --prop_exact PROP_EXACT
```

where NOISE_LV should be 0.01, 0.05 or 0.10 and PROP_EXACT 0.01,0.02 or 0.04.

In Figure 5, we report the analytical results (where we considered 10 exact data for T, here without any noise).

We fix both σ_D and σ_R and run the training for $n_{exact} = 10, 20, 40$ and noise_level $\epsilon = 0.01, 0.05, 0.10$. The noise level in this experiment is quite large (in particular when $\epsilon = 0.10$) compared to the range of T, that in this case goes from 0 to 0.86, with a mean close to 0.5. This means that we are adding to the

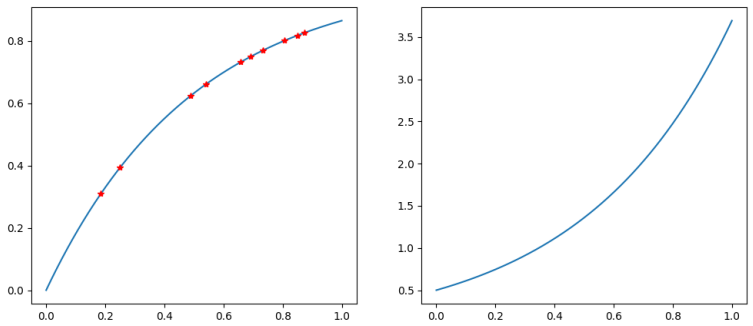


Figure 5: Activation times (with 10 exact data) and conduction velocity in 1D example

activation times a noise of 2%, 10% and 20%, respectively.
We compute the L^2 relative error and both mean and max standard deviation (in all the domain) for both T and v. Results are reported in table 1: the error decrease with a smaller noise and more data (and this is reasonable), while the uncertainty (very clearly for v, while not always true for T) increase with noise and decrease with more data.

n. data	noise	L^2 rel err (T,v)	ST mean (T,v)	ST max (T,v)
10	0.01	(0.012,0.099)	(0.0087,0.079)	(0.015,0.287)
10	0.05	(0.029,0.171)	(0.038,0.196)	(0.062,0.603)
10	0.10	(0.063,0.375)	(0.034,0.348)	(0.112,0.898)
20	0.01	(0.0086,0.086)	(0.013,0.052)	(0.016,0.164)
20	0.05	(0.034,0.065)	(0.021,0.105)	(0.035,0.281)
20	0.10	(0.068,0.218)	(0.017,0.123)	(0.027,0.374)
40	0.01	(0.0074,0.063)	(0.013,0.054)	(0.021,0.180)
40	0.05	(0.023,0.079)	(0.012,0.066)	(0.016,0.231)
40	0.10	(0.045,0.113)	(0.012,0.085)	(0.019,0.244)

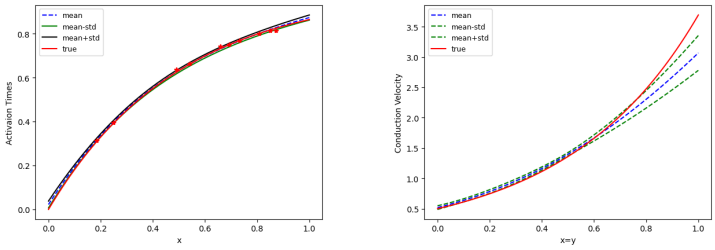
Table 1: 1D UQ analysis, n.data 10,20,40 and noise 0.01,0.05,0.10

We notice that the conduction velocity v has a bigger variability if we have few data, varying the noise, than if we have more data. With 10 exact values, the mean standard deviation on v goes from 0.079 (with a noise of 0.01) to 0.348 (with a noise of 0.10); with 20 data we start from 0.052 (with noise 0.01) up to 0.123 (with noise 0.10), while with 40 data we go from 0.054 (with noise 0.01) to 0.085 (with noise 0.10).
If we compute the ratio between $std^{n_{ex}}_{mean}(\epsilon = 0.10)$ and $std^{n_{ex}}_{mean}(\epsilon = 0.01)$ we

can find:

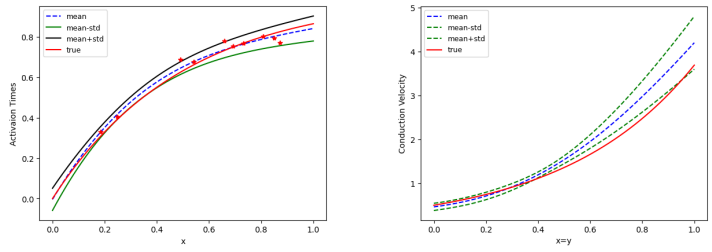
$$\begin{aligned}\frac{std^{10}_{mean}(\epsilon = 0.10)}{std^{10}_{mean}(\epsilon = 0.01)} &= 4.41 \\ \frac{std^{20}_{mean}(\epsilon = 0.10)}{std^{20}_{mean}(\epsilon = 0.01)} &= 2.37 \\ \frac{std^{40}_{mean}(\epsilon = 0.10)}{std^{40}_{mean}(\epsilon = 0.01)} &= 1.57\end{aligned} \tag{34}$$

hence it seems that there is a linear relation, with a doubling in number of exact data result in a half of the previous ratio. Of course these results show only a relationship of the ratio in standard deviation, but we can't be sure if we are underestimating or overestimating the uncertainty. Additional investigations are required to verify the capability of the model to quantify uncertainties in conduction velocity estimation.
In Figure 6 and 7 the first two cases of this analysis (10 data, noise 0.01 and 0.05).



(a) Activation times (b) Conduction velocity

Figure 6: Results with 10 exact data, noise level = 0.01



(a) Activation times (b) Conduction velocity

Figure 7: Results with 10 exact data, noise level = 0.05

7.2 1D with σ_D and σ_R trainable

If we consider also σ_D and σ_R as trainable, we can analyze the posterior distributions of these new hyperparameters with respect to the prior, varying for instance the noise level in the data. We expect that σ_D will increase with noise level, while σ_R should not be influenced by the noise on the exact data (while it depends on our noise on residual, that here we do not consider, since we consider $\mathbf{R} = \mathbf{0}$).

The following figures 8 and 9 confirm our belief: we can see that posterior distributions of σ_D increase with noise level, while they are similar for σ_R.

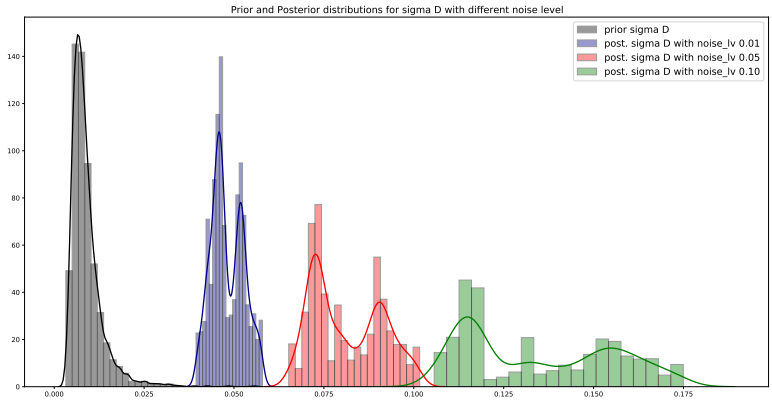


Figure 8: σ_D prior and posteriors for noise level = 0.01, 0.05, 0.10

7.3 2D Anisotropic UQ

The second experiment is a 2D Anisotropic Eikonal equation (3) in $(0,1)^2$ domain, with a source placed at $(x, y) = (0.5, 0.5)$. In this case, we know the analytical solution, that is:

$$\begin{aligned} T(x,y) =& 1 - \exp(-\sqrt{2(x-0.5)^2 + 2(x-0.5)(y-0.5) + (y-0.5)^2}) \\ a(x,y) =& \frac{1}{\exp(-2\sqrt{2(x-0.5)^2 + 2(x-0.5)(y-0.5) + (y-0.5)^2})} \\ b(x,y) =& \frac{2}{\exp(-2\sqrt{2(x-0.5)^2 + 2(x-0.5)(y-0.5) + (y-0.5)^2})} \\ c(x,y) =& \frac{1}{\exp(-2\sqrt{2(x-0.5)^2 + 2(x-0.5)(y-0.5) + (y-0.5)^2})} \end{aligned} \tag{35}$$

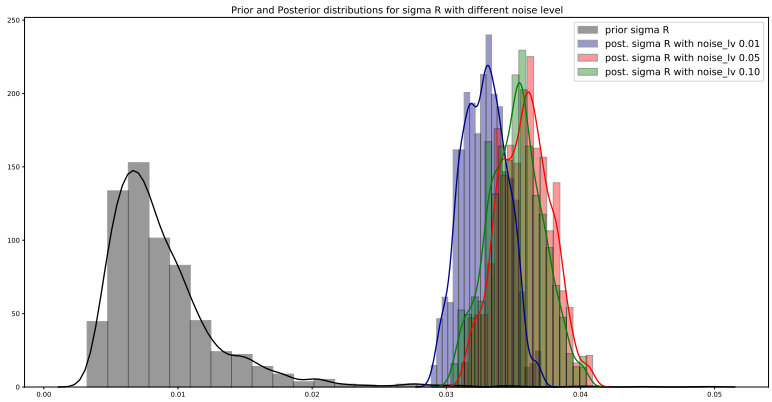


Figure 9: σ_R prior and posteriors for noise level = 0.01, 0.05, 0.10

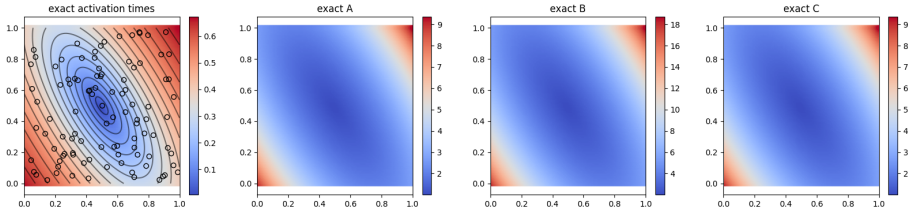


Figure 10: Exact result for T, a, b and c in 2D-Anisotropic example

with the conduction velocity tensor, as shown in (2), given by:

$$\mathbf{M}^2(\mathbf{x}) = \begin{bmatrix} a(x,y) & -c(x,y) \\ -c(x,y) & b(x,y) \end{bmatrix}.$$

Figure 10 shows the plot of exact solutions.
Since here we are in an anisotropic case, we have to equip the code also with a relation between the entries of the matrix $\mathbf{M}$ to make the problem solvable: indeed we can always find a linear combination between a,b and c that can be a solution.

$$\begin{cases} b(x,y) = 2a(x,y) \\ c(x,y) = a(x,y) \end{cases} \tag{36}$$

Here we work under the simplifying assumption of homogeneous coefficients. We impose these constraints in a "weak" sense (instead of a "strong" sense that could be outputs only $a(x,y)$ and then compute b and c using the relations), imposing the residual of both the relations in our likelihood of residual.
Figures 11 and 12 show the results for $T(x,y)$ and $a(x,y)$ with just 100 exact

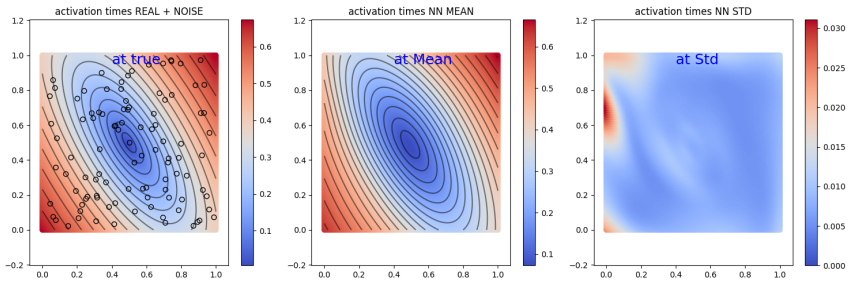


Figure 11: Activation times in 2D-Anisotropic, 100 data and 0.05 noise

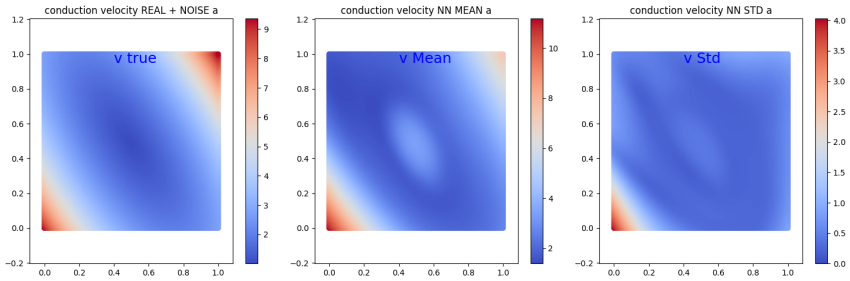
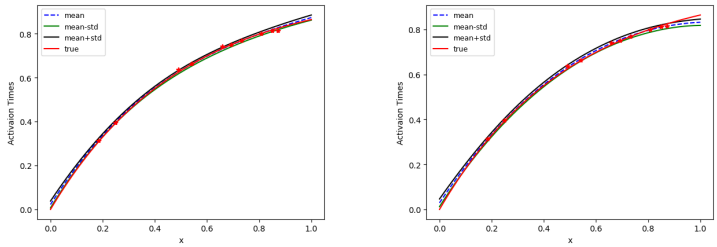


Figure 12: a(x,y) in 2D-Anisotropic, 100 data and 0.05 noise

data, affected by a noise level equal to 0.05. We can remark that the reconstruction of $a(x, y)$ (and similarly for $b(x, y)$ and $c(x, y)$) is affected by the complexity, in particular close to the boundary: the anisotropic problem is more difficult than the isotropic ones because of the relations between the three components of M.

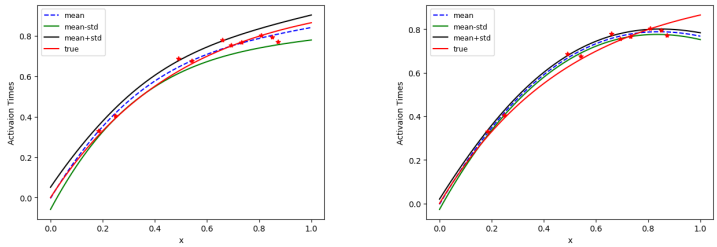
7.4 Influence of Physics-Informed (Eikonal) on errors and uncertainty in Activation Times

Knowing a physical model in these methods also work as a regularization technique in addition to the possibility of reconstructing the conduction velocity, it helps us to build a reasonable activation map even if we have a very small dataset of noisy measurements of activation time.
To see this, we have tried to run our code without the "Physics-Informed" part, to estimate just T, with a noisy and sparse dataset. We have performed the same training as in the 1D example, with just 10 data, but this time we have set $\alpha_{pde} = 0$, so our log posterior is build only by two components: log prior of theta and log likelihood of activation times data. The results in figure 13, 14 and 15 show that knowing the physics behind the phenomena also helps in reconstructing T substantially, since the cases "With Eikonal" show always better results than the "Without Eikonal" ones (and the difference increase with noise



(a) With Eikonal (b) Without Eikonal

Figure 13: Comparison of activation times with 10 exact data, noise_lv = 0.01



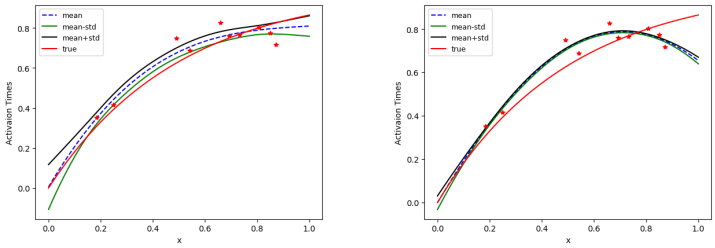
(a) With Eikonal (b) Without Eikonal

Figure 14: Comparison of activation times with 10 exact data, noise_lv = 0.05

level).

7.5 3D Case

We have tested the code also on a 3D case: an Isotropic case, for which the Forward Problem was solved with Pykonal (see the Appendix for more informations), on a prolate geometry (that is, a hemisphere with a small thickness). We have initialized the forward problem by setting a source local at the apex of prolate, and providing a conduction velocity of 0.5 in the half below, 1.0 in the upper half. The real activation times now range from 0 to more than 25. Here we consider a noise level of 0.5, 1.0 and 2.0. We use 80'000 collocation points and 400/800 exact data and an architecture of 5 Hidden layers, 100 neurons each. In figures 16 and 17 the results with 400 data and noise level 1.0, while in figures 18 and 19 with 800 data and same noise. We can clearly see that with more data we can achieve a better results. In Figure 20 we have displayed the relation between uncertainty (in terms of standard deviation) in activation times and localization of exact data: we can appreciate the fact that the uncertainty seems bigger where we have not enough data in the neighborhood. (All these 3D plots are made using an additional libraries, outside the code, called Mayavi, `https://docs.enthought.com/mayavi/mayavi/`).



(a) With Eikonal (b) Without Eikonal

Figure 15: Comparison of activation times with 10 exact data, noise_lv = 0.10

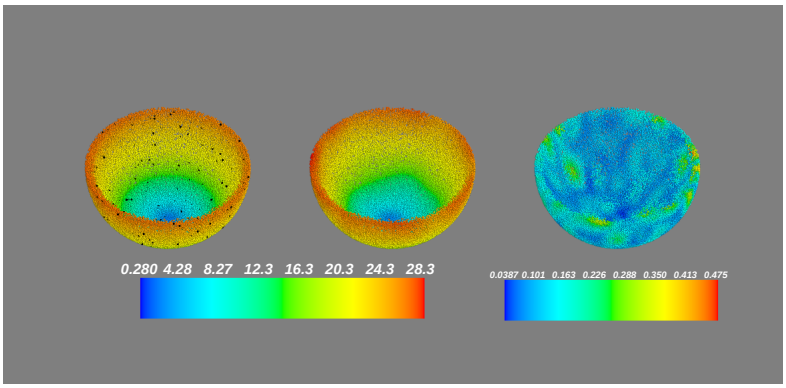


Figure 16: Activation times with 400 exact values

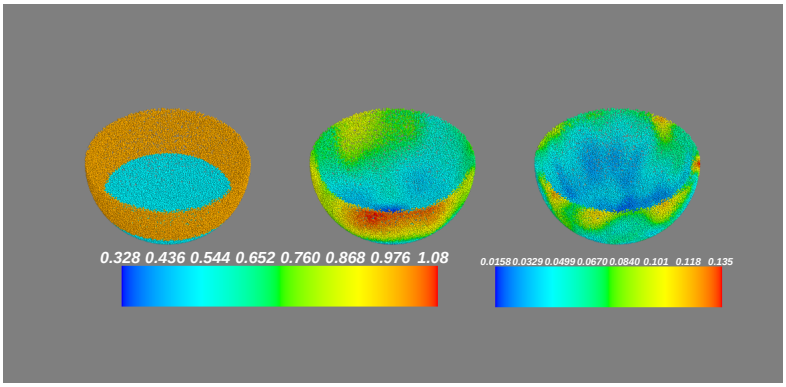


Figure 17: Conduction velocity with 400 exact values

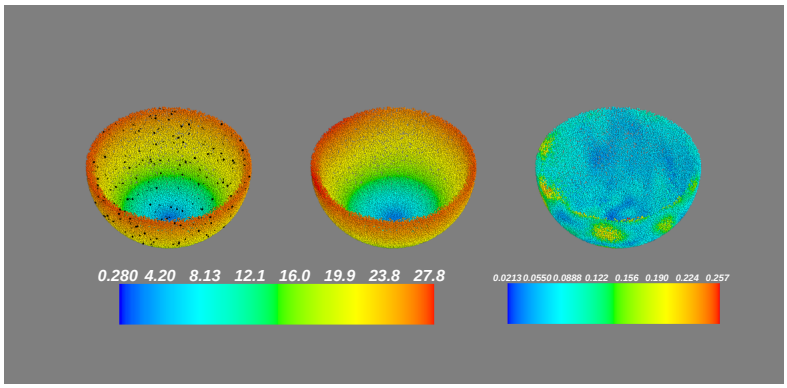


Figure 18: Activation times with 800 exact values

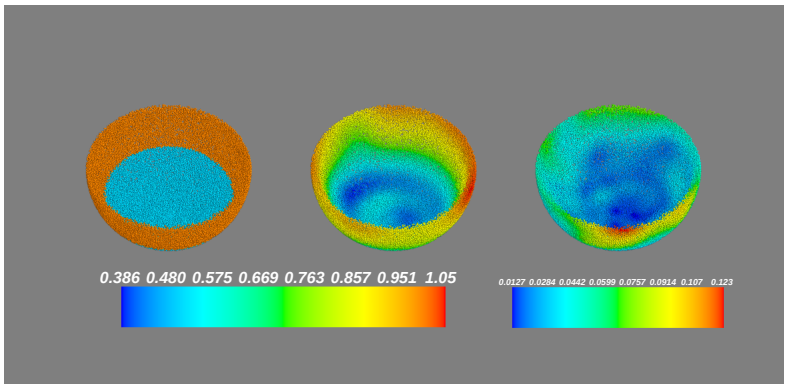


Figure 19: Conduction velocity with 800 exact values

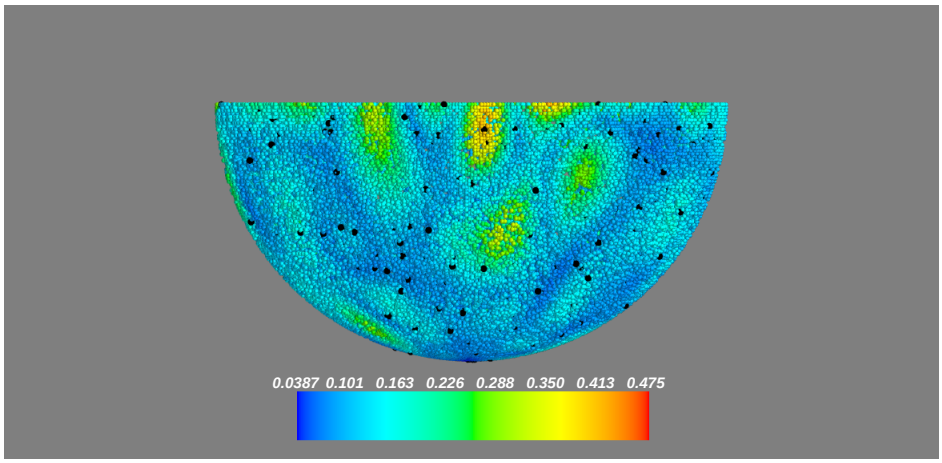


Figure 20: Uncertainty and exact data positions in Activation times

7.6 Comparison with SVGD

Using SVGD algorithm, we found good results in terms of error in reconstructing T and v, but weaker result in terms of Uncertainty Quantification. We compare the finding of 1D-case, with 10 exact data, noise level 0.01 and 0.05, between HMC and SVGD. We use here 30 NNs to approximate the posterior: choosing this parameter is the most critical part of the algorithm, since we have a tradeoff of computational efficiency and accuracy. In addiction to this, having NNs stored in our "BayesNN_SVGD" class can cause a memory allocation problem: if we cannot rely on a RAM with sufficient storage space we could end up with a OOM (Out Of Memory) python error. (We could fix this problem changing a little bit our implementation: instead of storing 30 Neural Networks class, FCN, we can only store 30 different $\boldsymbol{\theta}$ vector of parameters, and use only one network. In that case, we have to update and change our parameters 30 times every iteration, and this will cause a lower performance in terms of computational time: we are going to trade computational speed for memory spaces).

Method	n. data	noise	L^2 (T,v)	std_{mean} (T,v)	std_{max} (T,v)
HMC	10	0.01	(0.012,0.099)	(0.0087,0.079)	(0.015,0.287)
HMC	10	0.05	(0.029,0.171)	(0.038,0.196)	(0.062,0.603)
SVGD	10	0.01	(0.034,0.066)	(0.0021,0.021)	(0.018,0.089)
SVGD	10	0.05	(0.0717,0.354)	(0.0078,0.139)	(0.055,0.489)

Table 2: 1D UQ analysis, n.data 10,20,40 and noise 0.01,0.05,0.10

From the results in table 2, we can conclude that the SVGD method underestimates more the standard deviations than HMC: this problem can be motivated by the small number of NNs, too few to compute effectively a numerical standard deviation. We could increase that number but, as said before, we could also occur in problems of memory allocation; also if using more NNs, we might increase computational time.

8 Conclusion

In this project we have built a comprehensive implementation of Bayesian PINNs to solve Inverse problems involving the for Eikonal Equation. This work shows great promises in both reconstructing a good conduction velocity tensor for both Isotropic and Anisotropic Eikonal equation, and performing a comprehensive Uncertainty Quantification. Even if HMC provides better performances, it requires a lot of work behind the scene to reach a stable convergence, and some minor parameters change (described in the code comments); on the other hand SVGD is more robust but slower, and has a bigger problem of Memory Allocation (storing 30 NNs at the same time).
Some future improvements of the code might involve the use of another MCMC algorithm (and it will be quite simple to extend the code in this way) and exploit the code to address a real and more challenging 3D cases, maybe considering also anisotropy along fiber direction.
Finally, we can create an "Active Learning" algorithm, that tells us where to sample the next data position, in order to dinamically minimize the uncertainty on conduction velocities and times. This could be relevant also from a clinical point of view: an algorithm that tells the clinicians where to take the next measurements, maybe during the acquisition process. In that case, we would need to face with the problem of clinical time, and we have to require that the code will be fast enough to give an answer to the clinician during the cardiac mapping procedure.

A Tensorflow 2

A.1 Graph execution and Eager execution

One of the big changes from Tensorflow version 1 to version 2 is the so called "Eager execution". In Tensorflow 1 you have to write all tensor operations you need and in the end call "tf.Session.run()" to finally run all the operations. So we needed to first instantiate a **Graph** for computation, and just in the end all the computations are actually executed. These operations make the code very difficult to debug, since we cannot know how our tensors are being transformed during computations, but just at the end. New Tensorflow 2 "Eager execution" instead computes operations immediately, and our operations on tensors return real tensors at the same time, instead of constructing a computational graph to be run later (see, e.g., `https://www.tensorflow.org/guide/eager`).
With Eager execution, Tensorflow 2 can be used as a normal Python library, since now the execution follows directly Python interpreter, we can use easily all Python debugging tools and we end up with more readable code.

A.2 @tf.function() decorator

While Eager execution is better than Graph execution for all the reasons listed above, Graph execution is faster. This is clear when training bigger models, with a quite large dataset and for enough epochs. The reason is simple: Graph execution spends a big amount of computational time in the first epoch to build actually computational graphs; but after the first time, we pass our data and our parameters to these graphs in a simpler and faster way than calling functions like in Eager execution. For these reason, it is a good practice to initally use Eager execution, to have a simpler code and easier to debug, but after all the developing periods, to switch to Graph execution to same computational time. For this purpose, Tensorflow 2 has introduced the decorator "@tf.function()", that can be placed before definition of a function and says to Tensorflow to use graph execution on it. In this project we use this decorator on every functions that use "tf.GradientTape", and the final speed-up we obtain is quite impressive.

A.3 Comparison

Figure 21 shows the comparison of Eager and Graph execution of 1D example with an architecture of 5 hidden layers, 100 neurons each. Eager execution use approximatively 2 sec for each epochs, while the Graph execution needs more time in the first epoch (10 seconds) but then just 0.5 second for each epochs; after some first epochs we can see clearly that Graph computation becomes more convenient.

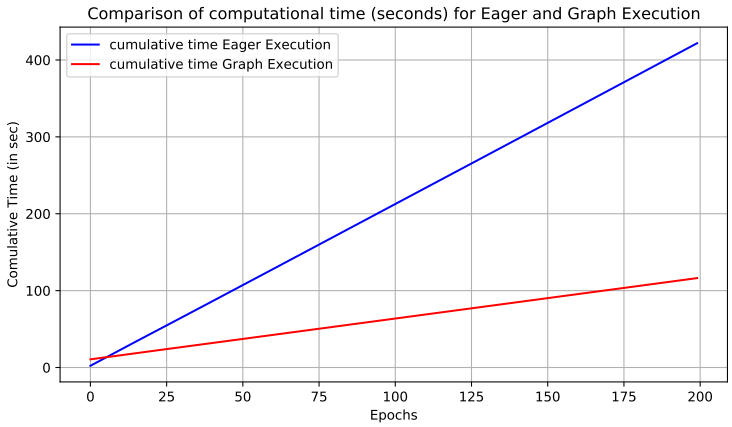


Figure 21: Computational time comparison for Eager and Graph computation

B GPU Accelerator

Since Neural Network Architectures are now becoming more and more complex, with the number of parameters that reaches hundred of millions or more, GPU computations becomes more and more important.
In this project we do not use bigger and complex network architectures, since they are just simple Fully Connected Networks with a small number of hidden layers (for instance 3,4 or 5) and a quite large number of neurons for each layer (from 30 to 100). The number of parameters is very smaller than other modern Neural Networks, ranging from 1000 to 50'000; therefore we can't see a big difference using GPU computation, because even if it is faster than a normal CPU computation in these types of task, it requires much more time to initialize all the computations (with respect to CPUs) and the difference in each epoch is considerable only for very big networks.
Below we report the results for a small Network (architecture: 3 hidden layers, 20 neurons each) and a "big" Network (5 hidden layers, 100 neurons) we have used in the 1D Isotropic UQ example. The devices we have used are:

- **CPU**: Intel Core i7-8550U 1.8GHz, with Turbo Boost up to 4.0GHz
- **GPU**: NVIDIA GeForce MX130 2GB VRAM.

In figures 22 and 23 we show how in a small network CPU computing is also faster than GPU. This is due to the fact that with a small parameter dimension GPU is not necessary, because the initialization time might be higher than the actual computational time. We can understand this fact by looking at the results obtained for a bigger network, where now GPU computing is convenient: the computational time with a CPU on big network is 2.38 times the time spent by the CPU for small network (1092s / 459s), while for GPU computing the ratio

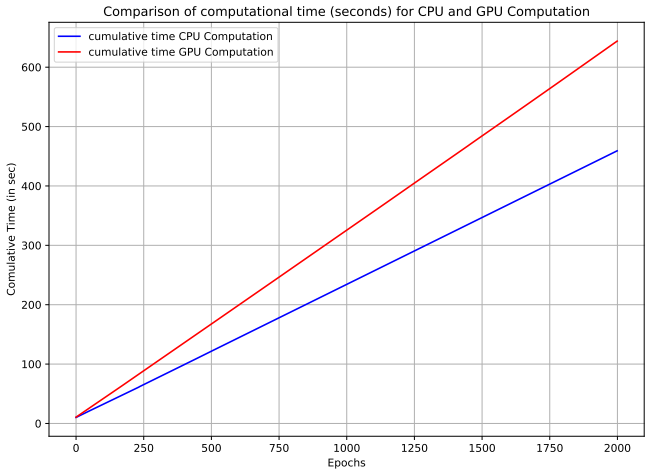


Figure 22: Computational time comparison for CPU and GPU in a small network (3/20 architecture)

is only 1.23 (794s / 643s). The difference will increase with bigger and bigger network architectures, for instance with an architecture made by 7 hidden layers with 500 neurons each, CPU computation took 6105s, 3777s in the case of GPU computations.

C Automatic Differentiation

With Automatic Differentiation (AD) [1] we indicate all the techniques to evaluate the derivative of a function by computer programs using the chain rule. The basic idea behind these methods is that a computer evaluates a function, providing an input, by a composition of basic functions that use elementary arithmetic operations and elementary functions.
Since the forward pass from input to compute the output is a chain of basic operations, we can compute derivatives using this chain backward, employing the chain rule for derivative, knowing only the basic derivatives (derivative of the sum, product by a constant, polynomials, exponential, trigonometric functions and so on). In this way, we can achieve better results than every other approximate numerical differentiation formula, since here we are computing the exact derivative.
These methods are widely used in every Machine Learning methods, and so also in Neural Networks, mostly for computing back-propagation derivatives of parameters $\boldsymbol{\theta}$, in according to loss functions. In this way, we can easily update our $\boldsymbol{\theta}$ after having collected $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})$ through automatic differentiation, using an optimizer like the Gradient Descent method and Adam optimizer.

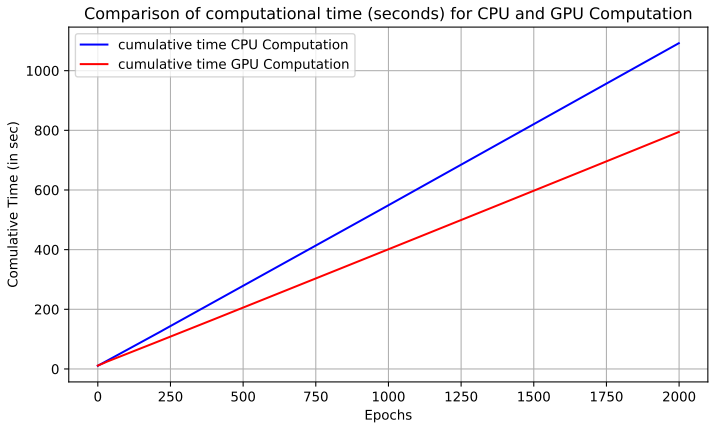


Figure 23: Computational time comparison for CPU and GPU in a big network (5/100 architecture)

In addition to this classical use of Automatic Differentiation, in this project (as well as in every work of the field "Scientific Machine Learning") we employ AD to compute derivatives of output with respect to input data (collocation points), not only with respect to parameters. This allows us to compute for instance $\frac{\partial T}{\partial x}$, $\frac{\partial T}{\partial y}$ and $\frac{\partial T}{\partial z}$, in order to have $\nabla T(\mathbf{x})$ and compute the residual of the Eikonal equation. Let us analyze more in detail how this works in our case.
In order to keep everything simple, suppose we are in the case of 1D isotropic Eikonal quation, with only 2 hidden layers, each of them having 3 neurons. Figure 24 shows the resulting architecture, where for completeness we have drawn also the bias vectors (in yellow).

After our 1D input (that is simply the first neuron x), in the first layer we have our weights matrix $\mathbf{W}^{(1)}$ and bias vector $\mathbf{b}^{(1)} = w_0^1$, the same in the second with $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)} = w_0^2$, and finally our output $\hat{y}_1 = T$ and $\hat{y}_2 = v$. The simple chain of computations that goes from input x to output (T, v) is therefore:

$$\begin{aligned} \mathbf{a_1} &= \mathbf{W}^{(1)} x + \mathbf{b}^{(1)}, \\ \mathbf{z_1} &= \sigma^{(1)}(\mathbf{a_1}), \\ \mathbf{a_2} &= \mathbf{W}^{(2)} \mathbf{z_1} + \mathbf{b}^{(2)}, \\ (T, v) &= (a_{2,1}, a_{2,2}), \end{aligned} \tag{37}$$

where σ^l are our activation functions, that in this case are equal to the Swish activation function

$$\sigma^{(l)}(x) = x \; sigmoid(x) = \frac{x}{1 + e^{-x}} \quad \forall l = 1, \ldots, L-1$$

and equal to identity function for $l = L$. Using the chain rule to compute the

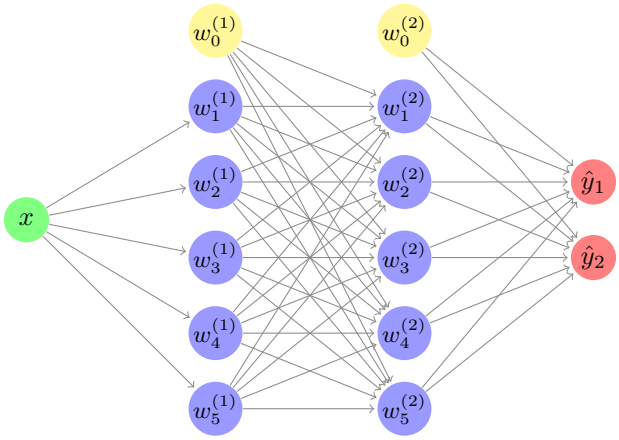


Figure 24: Architecture in 1D isotropic Eikonal

derivative is simple, indeed:

$$\frac{\partial T}{\partial x} = \frac{\partial a_{2,1}}{\partial \mathbf{z_1}} \frac{\partial \mathbf{z_1}}{\partial \mathbf{a_1}} \frac{\partial \mathbf{a_1}}{\partial x} = \mathbf{W_1^{(2)}} \partial \sigma^{(1)}(\mathbf{a_1}) \mathbf{W^{(1)}}, \tag{38}$$

and knowing the exact form of derivative of swish activation function:

$$\partial \sigma(x) = \sigma(x) + sigmoid(x)(1 - \sigma(x))$$

we have computed the derivative of T with respect to x directly using the architecture of the network.

D PyKonal library

To solve the forward problem in those cases where we have no analytical solution, like in the 3D example showed, we rely on the library Pykonal [5], a python library solving the forward isotropic Eikonal problem.
Pykonal implements the Fast Marching Method [11] in two or three dimensions, using both Cartesian and Spherical coordinates, to solve Isotropic Eikonal equation (6). After choosing which domain to use,a velocity field and source positions must be provided.
All the implementation of PyKonal can be found in his Github repository `https://github.com/malcolmw/pykonal`; starting from Examples showed in `https://malcolmw.github.io/pykonal-docs/examples/examples.html` and with some changes you can collect all the datasets we have used in our project.

References

[1] Atilim Baydin, Barak Pearlmutter, Alexey Radul, and Jeffrey Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18:1–43, 04 2018.

[2] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv preprint arXiv:1701.02434*, 2018.

[3] Ender Konukoglu, Jatin Relan, Ulas Cilingir, Bjoern Menze, Phani Chinchapatnam, Amir Jadidi, Hubert Cochet, Mélèze Hocini, Hervé Delingette, Pierre Jais, Michel Haïssaguerre, Nicholas Ayache, and Maxime Sermesant. Efficient probabilistic model personalization integrating uncertainty on data and parameters: Application to eikonal-diffusion models in cardiac electrophysiology. *Progress in biophysics and molecular biology*, 107:134–46, 07 2011.

[4] Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm. *arXiv preprint arXiv:1608.04471*, 2019.

[5] Nori Nakata Yehuda Ben-Zion Malcolm C. A. White, Hongjian Fang. Pykonal: A python package for solving the eikonal equation in spherical and cartesian coordinates using the fast marching method. *Seismological Research Letters*.

[6] M.T. Musavi, W. Ahmed, K.H. Chan, K.B. Faris, and D.M. Hummels. On the training of radial basis function classifiers. *Neural Networks*, 5(4):595–603, 1992.

[7] R. L. Nelson and H. Kwan. "leapfrog" methods for numerical solution of differential equations, sinewave generation, and magnitude approximation. 2:802–806 vol.2, 1995.

[8] Alfio Quarteroni, Andrea Manzoni, and Christian Vergara. The cardiovascular system: Mathematical modelling, numerical algorithms and clinical applications. *Acta Numerica*, 26:365–590, 05 2017.

[9] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686 – 707, 2019.

[10] Francisco Sahli Costabal, Yibo Yang, Paris Perdikaris, Daniel E. Hurtado, and Ellen Kuhl. Physics-informed neural networks for cardiac activation mapping. *Frontiers in Physics*, 8:42, 2020.

[11] J A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[12] Luning Sun and Jian-Xun Wang. Physics-constrained bayesian neural network for fluid flow reconstruction with sparse and noisy data. *arXiv preprint arXiv:2001.05542*, 2020.

[13] Liu Yang, Xuhui Meng, and George Em Karniadakis. B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data. *Journal of Computational Physics*, 425:109913, 2021.