POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Report

Uncertainty Quantification in Scientific Machine Learning

Numerical Analysis for Partial Differential Equations

Authors: Andrea Bonifacio

Academic year: 2022-2023

---

1. Introduction

This report aims to be a recap on Uncertainty Quantification in Physics Informed Neural Networks. Most of the work present here comes from [3].
The brief introduction about Neural Networks (NNs) and Uncertainty Quantification (UQ) comes from [1].

1.1. Neural Networks

Today NNs are all over the news, they are the tools behind many new inventions, like large language models. Here I'll describe the simplest possible architecture for a neural network.
In its most basic configuration, a feed-forward neural network is made up of three layers:

- Input layer,
- Hidden layer,
- Output layer.

While the first and last layers are quite self-explanatory (the input layer takes a vector $\boldsymbol{x}$ D-dimensional as an input and the output layer returns an object compatible with the dimensions of the dataset), the hidden layer is made up of multiple units, called neurons, which are basically vector transformations, to which is applied a nonlinear function. The general idea behind is to minimize the loss between the predictions of the model and the real values of the training dataset, by continuously modifying the neurons. Once a satisfying loss between the prediction and real results is reached, the model is ready to be used.
One of the main features of NNs is that they work really well as function approximators, so one of the logical step after their introduction was to use them to solve PDEs [2], but, at the time, it didn't lead anywhere until recently, when the advent of new tools and machine learning frameworks made possible to work again on solving PDEs using neural networks built up for the task.

1.2. Uncertainty Quantification

To describe uncertainty in a model, we must distinguish between the two different kinds. The model itself cannot represent reality as a whole, it will always be built up on assumptions that adds up creating epistemic uncertainty, which must be acknowledged, as it is irreducible. In the real world, data are collected by humans and sensors, which are both prone to errors and noise. This is the second kind of uncertainty, called aleatoric. To sum up, the total uncertainty in a model is made of two components:

$$\text{TU} = \text{EU} + \text{AU}.$$

To deal with epistemic uncertainty, it is possible to use a probability distribution over the model parameters. Given a dataset $D = \{\boldsymbol{x}_i, y_i\}_{i=1}^N$ where $\boldsymbol{x}_i \in \mathbb{R}^D$ are the inputs and $y_i \in \{1, \dots, C\}$ are the corresponding outputs in C different classes. The idea is to optimize the parameters ω of a function $y = f^\omega(\boldsymbol{x})$. The Bayesian

approach defines a model likelihood $p(y|\boldsymbol{x}, \omega)$, from which we can obtain the posterior distribution for a given dataset, thanks to Bayes' theorem:

$$p(\omega|X, Y) = \frac{p(Y|X, \omega)p(\omega)}{p(Y|X)}.$$

Now, given a test sample $\boldsymbol{x}^*$, a class label can be predicted as

$$p(y^*|\boldsymbol{x}^*, X, Y) = \int p(y^*|\boldsymbol{x}^*, \omega)p(\omega|X, Y)\, d\omega.$$

2. Problem setup

The problem can be formulated in the following way

$$\begin{cases} \mathcal{N}_x\left[u(x;\omega); k(x;\omega)\right], & x \in \mathcal{D}, \omega \in \Omega, \\ \mathcal{B}_x\left[u(x;\omega)\right] = 0, & x \in \Gamma, \end{cases} \tag{1}$$

where $\mathcal{N}_x$ is a differential operator, $\mathcal{D}$ the domain, Ω the random space and $u(x;\omega)$ the solution. On the boundary Γ we have the boundary conditions imposed by the operator $\mathcal{B}_x$. $k(x;\omega)$ represent the random parameter.
There are two possible problems:

- Forward problems: in this case, we know the distribution of $k(x;\omega)$ everywhere on the domain, and the quantity of interest is $u(x;\omega)$.
- Inverse problems: when the information about $k(x;\omega)$ is partially known, but there is a lot more information about $u(x;\omega)$ it is possible to infer the full distribution of k.

3. Methodology

3.1. PINNs for deterministic systems

Here is how to solve differential equations using PINNs. To do so, (1) must be rewritten replacing the random inputs with a set of finite parameters, to transform it in a deterministic problem.

$$\begin{cases} \mathcal{N}_x\left[u;\eta\right], & x \in \mathcal{D}, \\ \mathcal{B}_x\left[u\right] = 0, & x \in \Gamma. \end{cases} \tag{2}$$

The neural network will be denoted by $\hat{u}(x;\theta)$, which is the approximation of the solution $u(x)$ with a specific set of parameters θ dependent on the network. In a classical deep learning setting, the network should have one constraint, which is to reproduce the values in the dataset. In a physics informed settings, there is a second constraint, which is that the network should comply with the physical laws imposed by (2).

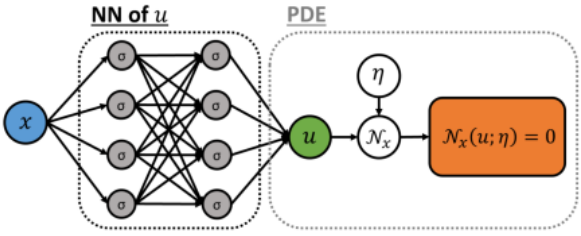

Figure 1: Schematic of a PINN for solving differential equations. Figure from [3].

To do so, a second network is defined

$$\hat{f}(x;\theta,\eta) := \mathcal{N}_x\left[\hat{u}(x;\theta);\eta\right], \tag{3}$$

which is computed straightforwardly from $\hat{u}$. The parameters of the second network are the same of the first one. Assuming a dataset with N_u observations on u collected at $\{x_u^{(i)}\}_{i=1}^{N_u}$ and N_c the number of collocation point in which $\hat{f}$ will be evaluated.

Algorithm 1 PINN for solving differential equations

Step 1: Specify the training set

$$\hat{u} : \{(x_u^{(i)}, u(x_u^{(i)}))\}_{i=1}^{N_u}, \quad \hat{f} = \{x_f^{(i)}, 0\}_{i=1}^{N_c}.$$

Step 2: Construct a NN $\hat{u}(x;\theta)$ with random initialized parameters θ.
Step 3: By using automatic differentiation, construct the residual network $\hat{f}(x;\theta,\eta)$.
Step 4: Specify a loss function that includes both networks

$$\mathcal{L}(\theta,\eta) = \frac{1}{N_u}\sum_{i=1}^{N_u}[\hat{u}(x_u^{(i)};\theta) - u(x_u^{(i)})]^2 + \frac{1}{N_c}\sum_{i=1}^{N_c}\hat{f}(x_f^{(i)};\theta,\eta)^2. \tag{4}$$

Step 5: Train the networks to find the best parameters by minimizing the loss function:

$$\theta = \arg\min \mathcal{L}(\theta,\eta)$$

3.2. Moving to a stochastic setting

The idea now is to analyze a stochastic problem, which brings back (1). A new dataset is needed to collect all the random events. This dataset will be made up of N_k measurement at $x_k^{(i)}$ locations in the domain. Given N measurements, the random instance at the s^{th} event will be called ω_s. Now, defining $k_s^{(i)} = k(x_k^{(i)};\omega_s)$ and $u_s^{(i)} = u(x_u^{(i)};\omega_s)$, the new dataset is built up like this:

$$\mathcal{S}_t = \{\{(x_k^{(i)}, k_s^{(i)})\}_{i=1}^{N_k}, \{(x_u^{(i)}, u_s^{(i)})\}_{i=1}^{Nu}, \{(x_f^{(i)}, 0)\}_{i=1}^{N_f}\}_{s=1}^{N}.$$

The next step is the arbitrary polynomial chaos (aPC) expansion, which will require:

1. Dimension reduction;
2. Constructing the aPC basis;
3. Building the NN-aPC network as a surrogate model and train the network for each mode.

3.2.1 Dimension reduction with PCA

To find a lower dimensional space, the proposed technique is principal component analysis (PCA), which finds a smaller number of modes able to explain almost all the dataset with enough accuracy. In this case, the reduced dataset will be the one about k. Let K be the $N_k \times N_k$ covariance matrix of the measurements on k

$$K_{i,j} = \text{Cov}(k^{(i)}, k^{(j)}).$$

Given λ_l and ϕ_l be the l-largest eigenvalue and its eigenvector, it is possible to write

$$K = \mathbf{\Phi}^T \mathbf{\Lambda} \mathbf{\Phi},$$

where $\mathbf{\Phi} = \phi_1, \phi_2 \ldots, \phi_{N_k}$ is an orthonormal matrix and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_{N_k})$ is a diagonal matrix. Let $\boldsymbol{k}_s = [k_s^1, k_s^{(2)} \ldots, k_s^{(N_k)}]^T$ be the results of the k measurement at the s^{th} moment. It is possible then to write

$$\boldsymbol{\xi}_s = \mathbf{\Phi}^T \sqrt{\mathbf{\Lambda}}^{-1} \boldsymbol{k}_s,$$

which is an uncorrelated random vector. Thanks to that, and the fact that $\boldsymbol{k}_0 = \mathbb{E}[\boldsymbol{k}]$ is the mean of each measurement, is it possible to rewrite

$$\boldsymbol{k}_s \approx \boldsymbol{k}_0 + \sqrt{\mathbf{\Lambda}^M} \mathbf{\Phi}^M \boldsymbol{\xi}_s^M, \quad M < N_k.$$

The choice of M depends on the accuracy that one wants to maintain in the low dimensional space.

3.2.2 Arbitrary polynomial chaos expansion

Assuming a set of M-dimensional samples of random vectors

$$S := \{\boldsymbol{\xi}_s\}_{s=1}^{N}$$

with hidden probability measure $\rho(\boldsymbol{\xi})$. With a sufficiently large number of observation, it is possible to approximate this probability measure with a discrete one

$$\rho(\boldsymbol{\xi}) \approx \nu_S(\boldsymbol{\xi}) = \frac{1}{N} \sum_{\boldsymbol{\xi}_s \in S} \delta_{\boldsymbol{\xi}_s}(\boldsymbol{\xi}),$$

where $\delta_{\boldsymbol{\xi}_s}$ is the Dirac measure. Then a set of $P+1$ multivariate orthonormal polynomial basis $\{\psi_\alpha(\boldsymbol{\xi})\}_{\alpha=0}^{P}$ can be constructed using Gram-Schmidt algorithm.

3.2.3 Learning stochastic modes

Now the networks must be trained to predict the stochastic modes of the QoI. The focus will be on the inverse problem, where both u and k are partially unknown. The first network $\hat{u_\alpha}$ takes the spatial coordinates x as input and returns a $(P+1) \times 1$ vector of the aPC modes of u evaluated in x, while the $\hat{k}$ network takes x as input and outputs a $(M+1) \times 1$ vector of the k modes. Thanks to that, u and k can be approximated as

$$\tilde{k}(x;\omega_s) = \hat{k}_0(x) + \sum_{i=1}^{M} \sqrt{\lambda_i}\hat{k}_i(x)\xi_{s,i},$$

$$\tilde{u}(x;\omega_s) = \sum_{\alpha=0}^{P} \hat{u_\alpha}(x)\psi_\alpha(\boldsymbol{\xi}_s).$$

The residual network is then obtained through automatic differentiation. Thanks to the aPC, it is possible to build a network that avoid being an uninterpretable black box. The idea is to build smaller networks which learns the various modes of $\hat{k}$ and $\hat{u}$. One network will learn the mean, one the first order basis and so on.

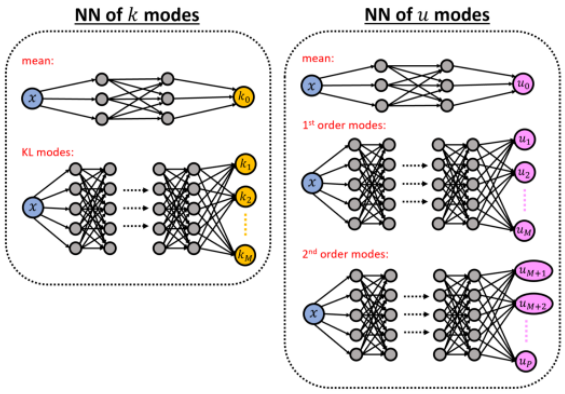

Figure 2: Schematic of the various networks needed for learning the stochastic modes. There are two smaller networks which calculates the two means and the bigger networks calculate the rest of the modes. Figure from [3].

The loss function is defined as the sum of the squared errors

$$\mathcal{L}(\mathcal{S}_t) = MSE_u + MSE_k + MSE_f,$$

where

$$MSE_u = \frac{1}{NN_u}\sum_{s=1}^{N}\sum_{i=1}^{N_u}\left[(\tilde{u}(x_u^{(i)};\omega_s) - u(x_u^{(i)};\omega_s))^2\right],$$

$$MSE_k = \frac{1}{NN_k}\sum_{s=1}^{N}\sum_{i=1}^{N_k}\left[(\tilde{k}(x_k^{(i)};\omega_s) - k(x_k^{(i)};\omega_s))^2\right],$$

$$MSE_f = \frac{1}{NN_f}\sum_{s=1}^{N}\sum_{i=1}^{N_f}\left[\left(\mathcal{N}_x[\tilde{u}(x_f^{(i)};\omega_s);\tilde{k}(x_f^{(i)};\omega_s)]\right)^2\right].$$

3.2.4 Dropout

The main problem with standard NNs is that they are not able to quantify model uncertainty. One common way to overcome this problem is the use of dropout. In the deep learning world, dropout is a technique used during the training of neural networks in which some neurons are randomly shut off during the training phase. This is made to prevent excessive co-tuning and to help the information spread across the network. In a classical setting, neurons are shut off only during the training phase, while for uncertainty quantification, they are shut off also at the test phase. Then the prediction y of the neural network can be estimated with the Monte Carlo (MC) method. Given T runs

$$\mathbb{E}(y) \approx \frac{1}{T}\sum_{t=1}^{T}\mathcal{NN}_t(x),$$

where $\mathcal{NN}_t$ is the t^{th} run with some dropped neurons.

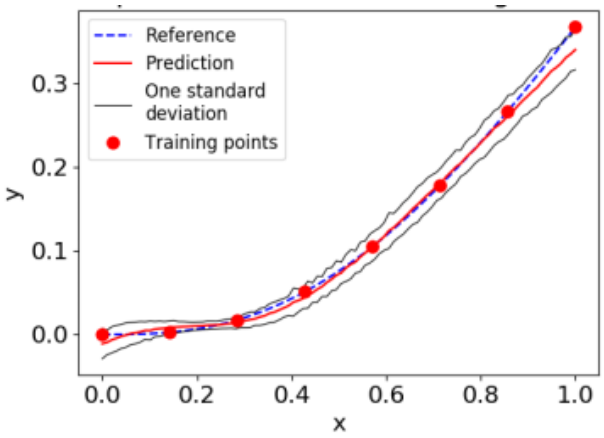

Figure 3: An example of using dropout for quantify uncertainty. Here the function $y = x^3e^{-x}$ is approximated using a NN. Mean and standard deviation are calculated from 1000 MC runs. Figure from [3].

4. Numerical Results

4.1. Solving SPDEs

The problem given involves a stochastic elliptic equation that reads

$$\begin{cases} -\dfrac{d}{dx}\left(k(x;w)\dfrac{d}{dx}u\right) = f(x) \quad x \in [-1,1], \omega \in \Omega \\ u(-1) = u(1) = 0 \end{cases} \tag{5}$$

In this case, there are some extra information about $u(x;\omega)$, but the distribution of the diffusion coefficient $k(x;\omega)$ is unknown. The forcing term $f(x) = 10$, while the randomness comes from $k(x;\omega)$. k it is modeled

and sampled from a non-Gaussian random process such that

$$\log(k(x;\omega)) \sim \mathcal{GP}(k_0(x), \mathrm{Cov}(x, x')),$$

where the mean $k_0(x) = \sin(3\pi x/2)/5$ and the covariance matrix has the form

$$\mathrm{Cov}(x, x') = \sigma^2 \exp\left(-\frac{(x-x')^2}{l_c^2}\right),$$

where $\sigma = 0.1$ and $l_c = 1.0$.
The dataset is built up using the MC sampling method. From a number $N = 1000$ of trajectories the data at the training points are extrapolated, for both u, k and f. Same for $N = 500$ trajectories that make up the test set. For the PCA, 6 variables are used, which are enough to explain 99% of the dataset, while for the aPC, a first order expansion is used.

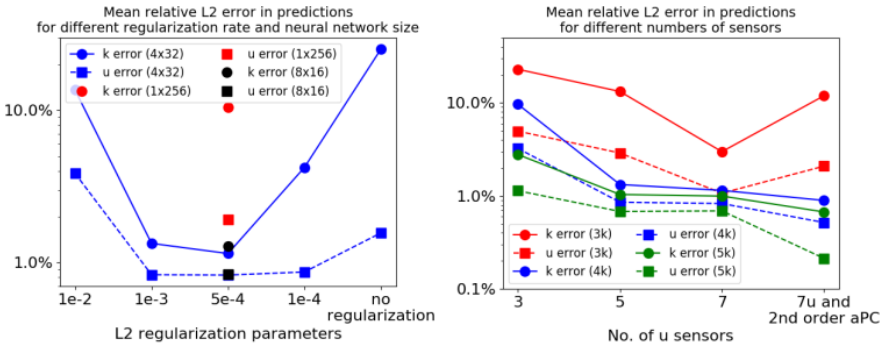

Figure 4: Comparing the prediction accuracy against various combinations of networks and values of the L2 regularizer λ (left side). Comparing the prediction accuracy against the number of training points (sensors) for u and k. Figure from [3].

Two groups of NNs are used, $\hat{k_i}$ and $\hat{u_\alpha}$, to calculate the modes of k and u. After some trial, as seen in Figure 4 with various shapes of networks, the optimal shape looks like to be 4 hidden layers with 32 neurons per layer, with a regularization parameter $\lambda = 0.0005$. Then, looking for the correct amount of training points, the balance is reached in 7 training points of u and 4 training points of k.

		mean	std	mode 1	mode 2	mode 3	mode 4
k	1st-order	0.54%	5.26%	4.15%	5.39%	12.03%	42.81%
	2nd-order	0.45%	1.87%	1.28%	1.95%	3.67%	29.95%
u	1st-order	0.14%	1.83%	0.98%	3.60%	4.34%	45.56%
	2nd-order	0.14%	0.51%	0.08%	0.80%	1.04%	32.16%

Figure 5: Comparing the L2 error when using 1^{st} and 2^{nd} order expansion aPC. Figure from [3].

As seen in Figure 5, the 2^{nd} order expansion aPC vastly improves the accuracy of the prediction. In fact, when testing both the 1^{st} and 2^{nd} order expansion is it possible to see how much better the 2^{nd} order expansion performs the task.

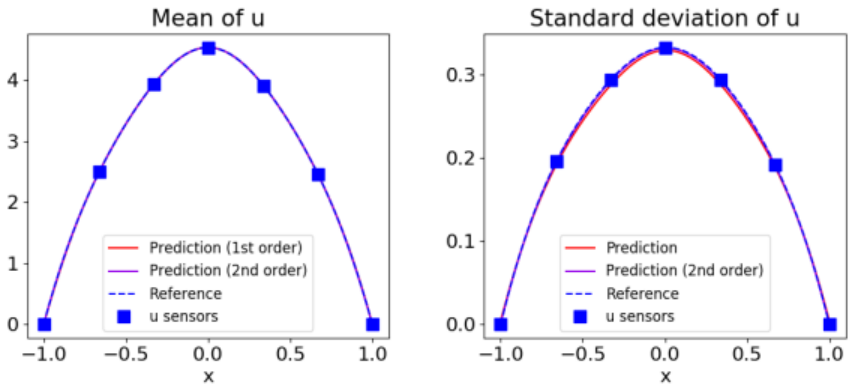

Figure 6: Predicted mean and standard deviation of u for various order expansions. Figure from [3].

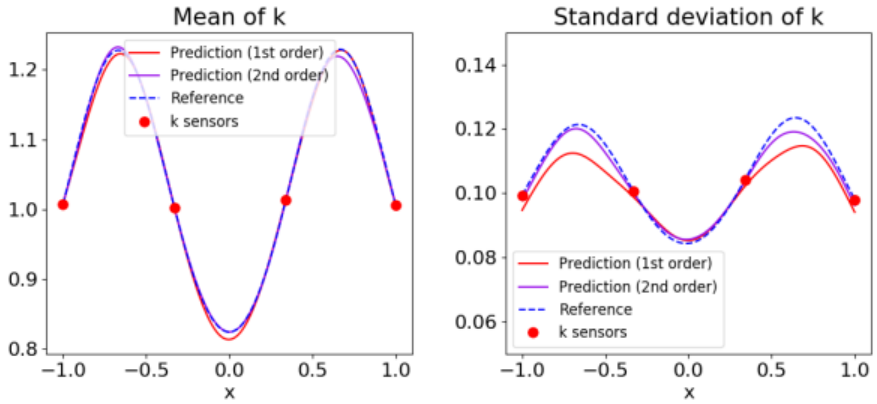

Figure 7: Predicted mean and standard deviation of k for various order expansion. Figure from [3].

Figure 6 shows how the networks performs in predicting the mean and standard deviation of u, in which is already clear that the higher order expansion is more accurate. In Figure 7 is shown that, despite having all the training points where the mean is 1.0 and the standard deviation is 0.1, the network is able to reconstruct the wavy structure of both, which is way more information that the data would provide.

4.2. Solving deterministic differential equations with PINNs and dropout

Is it possible, given a deterministic problem

5. Conclusions

References

[1] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U. Rajendra Acharya, Vladimir Makarenkov, and Saeid Nahavandi. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information Fusion*, 76:243–297, dec 2021.

[2] I.E. Lagaris, A. Likas, and D.I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, 1998.

[3] Dongkun Zhang, Lu Lu, Ling Guo, and George Em Karniadakis. Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. *Journal of Computational Physics*, 397:108850, nov 2019.