Deep Learning Methods for High-Dimensional Fluid Dynamics Problems: Application to Flood Modeling with Uncertainty Quantification

by

Pierre JACQUIER

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS, MECHANICAL ENGINEERING
M.A.Sc.

MONTREAL, MAY 8TH, 2020

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

M. Azzeddine Soulaïmani, memorandum supervisor
Génie Mécanique, École de Technologie Supérieure

M. Louis Lamarche, president of the board of examiners
Génie Mécanique, École de Technologie Supérieure

M. Gaétan Marceau Caron, external examiner
Applied Research Scientist, Mila

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON MAY 5TH, 2020

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my research advisor, Prof. Azzeddine Soulaïmani, for allowing me to work on such modern methods, and helping me throughout this research project. After exploring incredible yet not very practical methods for our case, he helped me decide to perform a big *pivot* at a crucial moment, and his reviews and insights on my writing have been incredibly helpful.

Dr. Azzedine Abdedou, my lab colleague, has also been very kind in explaining and sharing his work in the field of reduced-order modeling.

Vincent Delmas, friend and fellow graduate student, has helped me tremendously with his knowledge of clusters and his simulation data.

I want to thank the members of the jury, Prof. Louis Lamarche, and Dr. Gaétan Marceau Caron for the review and evaluation of this work.

My family has brought me massive emotional and financial support since day one in the university realm, and I can't be more grateful for the opportunities they've allowed and encouraged me to pursue.

Most of the heavy computations were performed on the vast infrastructure granted to researchers by Compute Canada and Calcul Québec, for which I'm very grateful.

Finally, the dual-degree opportunity that has been allowed by the open-mindedness of my home university, UTBM, is deeply appreciated, as well as the impressive administrative help from my UTBM supervisors, Prof. Dominique Chamoret and Prof. Sebastien Roth.

Méthodes d'apprentissage profond pour les problèmes de dynamique des fluides de haute dimension : application à la modélisation des inondations avec prise en compte des incertitudes

Pierre JACQUIER

RÉSUMÉ

Bien que des résultats impressionnants aient été obtenus dans les domaines bien connus où l'apprentissage approfondi a permis des percées telles que la vision par ordinateur, son impact sur des domaines différents et plus anciens reste encore largement inexploré. Dans le domaine de la mécanique des fluides numérique et en particulier dans la modélisation des inondations, de nombreux phénomènes sont de très haute dimension et les prévisions nécessitent l'utilisation de méthodes numériques, qui peuvent être, bien que très robustes et éprouvées, lourdes à calculer et peuvent ne pas s'avérer utiles dans le contexte de prévisions en temps réel. Cela a conduit à diverses tentatives de développement de techniques de modélisation à ordre réduit, à la fois intrusives et non intrusives. Une contribution récente nommée POD-NN consiste en la combinaison de la Proper Orthogonal Decomposition avec les réseaux neuronaux profonds. Pourtant, à notre connaissance, dans cet exemple et plus généralement sur le terrain, peu de travaux ont été menés sur la quantification des incertitudes émises par le modèle de substitution. Dans ce mémoire, nous visons à comparer différentes méthodes nouvelles traitant de la quantification de l'incertitude dans les modèles d'ordre réduit, en faisant avancer le concept POD-NN à l'aide de réseaux neuronaux informés des incertitudes, tels que les Deep Ensembles ou les Bayesian Neural Networks. Ces derniers sont testés sur des problèmes de référence, puis appliqués à une application réelle : les prévisions d'inondation dans la rivière des Mille-Îles à Laval, QC, Canada.
L'objectif est de construire un modèle de substitution non-intrusif, capable de *savoir quand il ne sait pas*, ce qui reste un domaine de recherche ouvert en ce qui concerne les réseaux neuronaux.

Mots-clés: Quantification de l'incertitude, apprentissage approfondi, POD espace-temps, modélisation des inondations

Deep Learning Methods for High-Dimensional Fluid Dynamics Problems: Application to Flood Modeling with Uncertainty Quantification

Pierre JACQUIER

ABSTRACT

While impressive results have been achieved in the well-known fields where Deep Learning allowed for breakthroughs such as computer vision, its impact on different older areas is still vastly unexplored. In Computational Fluid Dynamics and especially in Flood Modeling, many phenomena are very high-dimensional, and predictions require the use of numerical simulations, which can be, while very robust and tested, computationally heavy and may not prove useful in the context of real-time predictions. This issue led to various attempts at developing Reduced-Order Modeling techniques, both intrusive and non-intrusive. One recent relevant addition is a combination of Proper Orthogonal Decomposition with Deep Neural Networks (POD-NN). Yet, to our knowledge, little has been performed in implementing uncertainty-aware regression tools in the example of the POD-NN framework.

In this work, we aim at comparing different novel methods addressing uncertainty quantification in Neural Networks, pushing forward the POD-NN concept with Deep Ensembles and Bayesian Neural Networks, which we first test on benchmark problems, and then apply to a real-life application: flooding predictions in the Mille-Iles river in Laval, QC, Canada.

Building a non-intrusive surrogate model, able to *know when it doesn't know*, is still an open research area as far as neural networks are concerned.

Keywords: Uncertainty Quantification, Deep Learning, Space-Time POD, Flood Modeling

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

LIST OF ALGORITHMS

Page

LIST OF ABREVIATIONS

ETS	École de Technologie Supérieure
UTBM	Université de Technologie Belfort-Montbéliard
NN	Artificial Neural Network
DNN	Deep Neural Network
DNN	Bayesian Neural Network
GAN	Generative Adversarial Network
GP	Gaussian Process
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
AI	Artificial Intelligence
CFD	Computational Fluid Dynamics
POD	Proper Orthogonal Decomposition
SVD	Singular Value Decomposition
DOF	Degree of freedom
ROM	Reduced-Order Model
RB	Reduced Basis
LHS	Latin Hypercube Sampling
PINN	Physics-Informed Neural Network
UQPINN	Physics-Informed Neural Network handling Uncertainty Quantification

GP	Gaussian Process
NNGP	Neural-Net-induced Gaussian Processes
UQ	Uncertainty Quantification
PCE	Polynomial Chaos Expansion
POD-NN	Proper Orthogonal Decomposition framework involving Neural Networks
POD-EnsNN	Proper Orthogonal Decomposition framework involving Deep Ensembles
POD-BNN	Proper Orthogonal Decomposition framework involving Bayesian NNs
L-BGFS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno optimizer
MSE	Mean Squared Error
NLL	Negative Log-Likelihood
ReLU	Rectified Linear Unit
KL	Kullback-Leibler divergence
QC	Province of Quebec
1D	One-dimensional
2D	Two-dimensional
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SWE	Shallow Water equations
VI	Variational Inference
w.r.t.	with respect to

i.e.	*id est*, in other words
v.	versus

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

i, j, k	Indices
$\mathbb{R}$	Real space
$\mathcal{H}$	Hilbert space
log	Natural logarithm
Ω	Domain
Θ	Space domain in a vector space
L^2	Function space using the 2-norm
u	Function to be approximated by a surrogate model
$\hat{}$	Approximation symbol by a surrogate model
u_D	Function to be approximated by a surrogate model, with the spatial mesh built-in
u_{DB}	Function to be approximated by a surrogate model, with spatial mesh built-in and reduced
σ_D	Standard deviation around the predicted function $\hat{u}_D$
$\boldsymbol{U}$	Matrix of snapshots
$\boldsymbol{s}$	Non-spatial parameters to generate the matrix of snapshots
$\boldsymbol{V}$	POD modes/basis
$\boldsymbol{v}$	POD projection coefficients
$\boldsymbol{\varphi}^D$	Individual POD mode/basis
ψ	Basis function

ϵ	POD truncation criterion
$\boldsymbol{W}$	First matrix return by POD
$\boldsymbol{D}$	Matrix holding the singular values in POD
$\boldsymbol{Z}$	Matrix used in the generation of $\boldsymbol{V}$
$\boldsymbol{T}$	Contracted time-trajectory used in the two-step POD algorithm
ξ	Singular value
L	Number of truncated projection coefficients
n	Number of space dimensions
N_{x_i}	Number of points in an orthogonal mesh along the direction i
N_D	Number of nodes in a mesh
N_t	Number of time-steps
P	Number of non-spacial parameters of u
D	Size of the output of the function u
H	Total number of degrees of freedom
S	Total number of snapshots available from u
N_S	Number of non-spatial parameters snapshots
N_t	Number of time-steps
$\boldsymbol{X}$	Inputs of a Neural Network
$\boldsymbol{y}$	Outputs of a Neural Network
$\boldsymbol{w}$	Weights of a Neural Network

$\boldsymbol{b}$	Biases of a Neural Network
$\boldsymbol{h}^{(j)}$	Hidden layer j state of a Neural Network
d	Depth of a Neural Network, or degree of polynomial fitting function
λ	Regularization coefficient
$l^{(j)}$	With the layer j of a Neural Network
ϕ	Activation function of a Neural Network
N_e	Number of epochs (or iterations) for the training of a Neural Network
η	Leaning rate for the training of a Neural Network
N	Number of training points
$\mathcal{L}$	Loss function for the training of a Neural Network
RE	Relative error metric
$\mathcal{N}$	Normal distribution
p	Probability of a continuous variable
$\mathcal{U}$	Uniform distribution
$\mathbb{E}$	Expectation
$\mathcal{D}$	Dataset
$\mathcal{N}_x$	Nonlinear operator in a PDE setting
$\mathcal{L}_x$	Linear operator in a PDE setting
$\mathcal{R}$	Residual operator in a POD-Galerkin procedure
$\boldsymbol{I}$	Identitity matrix of context-infered size

$\mathbf{0}$	Null matrix of context-infered size
$_{\text{val}}$	Subscript denoting the validation dataset
$_{\text{tst}}$	Subscript denoting the testing dataset
$_{\text{out}}$	Subscript denoting out-of-distribution
$\mathcal{F}$	Variational free energy
$\boldsymbol{\epsilon}$	Random noise in variational inference
$\boldsymbol{\theta}$	Parameters of network or distribution (for VI)
N_{mc}	Number of samples for VI approximation
N_{ex}	Number of samples for expanded distribution approximation
N_{up}	Number of samples in uncertainty propagation
h	Depth of water
η	Elevation of the water surface
v_x	Velocity on the first axis of the fluid
v_y	Velocity on the second axis of the fluid
m	Manning roughness
g	Gravity density
$\boldsymbol{G}, \boldsymbol{H}, \boldsymbol{S}$	Matrices involved in SWEs
ζ	Adversarial training coefficient

INTRODUCTION

0.1 Background and the computational cost issue

While impressive results have been achieved in the well-known fields where Deep Learning allowed for breakthroughs such as computer vision, language modeling, or content generation, respectively performed in Szegedy, Ioffe, Vanhoucke & Alemi (2017), Mikolov, Sutskever, Chen, Corrado & Dean (2013), and Karras, Laine, Aittala, Hellsten, Lehtinen & Aila (2019), its impact on different, older fields is still vastly unexplored. In Computational Fluid Dynamics and especially in flood modeling, many phenomena are very high-dimensional, and predictions require the use of finite element or volume methods, which can be, while very robust and tested, computational-heavy and may not prove useful in the context of real-time predictions.

The world, as we know it today, is continuously evolving, and as the human concentration in large metropolitan areas keeps on increasing, it is of primary concern to focus on predicting natural disasters consequences. Montreal, QC, Canada, as many big metropolia, is located near large bodies of water, and their levels have to be monitored closely. The time for action after any variation remains ridiculously tiny compared to the importance of the measures that have to be taken to ensure public safety. However, with the many parameters involved and the computational cost of running simulation for free surface flows, especially in a large-scale, high-performance context, making real-time predictions and knowing the related arising uncertainties remain very challenging as of today.

0.2 Objectives

This led to various attempts at developing Reduced-Order Modeling techniques, both intrusive and non-intrusive, to develop surrogate, alternative models that could be used in a real-time context. A recent relevant addition is a combination of Proper Orthogonal Decomposition with artificial Neural Networks, first coined as POD-NN in Hesthaven & Ubbiali (2018), and extended to time-dependent problems in Wang, Hesthaven & Ray (2019). With its offline-online

paradigm, it allows for computational load offsetting to slower times as far as natural disasters are concerned, and provides rapid evaluation for new predictions in the urgency context. It has been applied to multiple physics applications, and we ultimately aim at using it for our flood modeling problem. The high nonlinear regression power that comes with Deep Neural Networks will be instrumental in tackling a wide range of real-world problems.

Nonetheless, to our knowledge, in this example and more generally in the field, little work has been conducted on quantifying uncertainties arising from the surrogate model, and this would represent a significant contribution.

In this work, we, therefore, aim at comparing different novel methods addressing uncertainty quantification in Machine Learning tools, especially in Deep Neural Networks, pushing forward the POD-NN concept with novel techniques like Deep Ensembles, Lakshminarayanan, Pritzel & Blundell (2017), or Bayesian Neural Networks, Blundell, Cornebise, Kavukcuoglu & Wierstra (2015). Chosen methods are tested on a variety of benchmark problems, and then deployed to a real-life application: flooding predictions in the Mille-Iles river in Laval, QC, Canada.

For the flood prediction application, our setup involves a set of input parameters resulting from on-site measurements. High-fidelity solutions are then generated using our own finite-volume code CuteFlow, which is solving the highly nonlinear Shallow Water equations. The goal is then to build a non-intrusive surrogate model that is able to *know when it doesn't know*, which is still an open research area as far as neural networks are concerned, Blundell *et al.* (2015); Lakshminarayanan *et al.* (2017).

We set out three main objectives for the surrogate model in this study:

1. Stay **general** enough so the framework can be used without too much change in various cases,

2. Be **fast** to predict the desired quantity for any new set of parameters, allowing for real-time predictions,

3. Give a **confidence interval** around the predictions, for it to be used out-of-distribution without making ridiculous claims.

0.3 Methodology outline

After reviewing the current state of our problem in the literature in Chapter 1, with a focus on the separation between intrusive and non-intrusive methods, fundamentals of Machine Learning and modern developments, a detailed description of our building block for this work, the POD-NN approach, is available in Chapter 2. A few possible strategies are discussed and compared in Chapter 3, investigating the different possibilities to fulfill our Objective 3.. Applying these uncertainties-handling techniques to the POD-NN framework in the flood modeling context is our contribution as a standalone research paper, detailed in Chapter 4.

CHAPTER 1

LITERATURE REVIEW

Computational Mechanics, and especially Fluid Dynamics, is an area of science that requires a lot of computing power to reach accurate outcomes. It almost always relies on a *mesh*, and its coarseness is directly related to the time needed for a simulation involving it to converge. It can become so large that ways to reduce its order have to be developed. These ways aim at constructing a Reduced-Order Model (ROM), that can effectively replace its heavier counterpart for tasks like design and optimization, or real-time predictions, all of which would require the model to run a large number of times, which is in most cases impossible by lack of adequate and available computer resources.

1.1 Reduced-basis through Proper Orthogonal Decomposition

The most common way to build a ROM is to go through a compression phase into a *reduced space*, defined by a set of Reduced Basis (RB), which is at the root of many methods, according to Benner, Gugercin & Willcox (2015). For the most part, RB methods involve an *offline-online* paradigm, where the first is the more computational-heavy one, while the latter should be fast enough to allow for real-time predictions. The idea is to collect data points from simulation, or any high-fidelity source, called *snapshots* and stored in an ensemble $\{\boldsymbol{u}^i\}$, and extract the information that has the broader impact on the dynamics of the system, the *modes*, via a reduction method in the *offline* stage.

Proper Orthogonal Decomposition was introduced in Lumley then presented in Holmes, Lumley, Berkooz, Mattingly & Wittenberg (1997) and Sarkar & Ghanem (2002), and aims at finding a basis functions $\varphi^{(k)}$ in a Hilbert space $\mathcal{H}$ possessing the structure of an inner product $(\cdot, \cdot)$, that would optimally represent the field u. Supposing that solutions or high-fidelity measures of theses solutions are availables as $\{\boldsymbol{u}^i\}$, each belonging to the same space $\mathcal{H}$, so that the field can be approximated in

$$u = \sum_{k=1}^{\infty} v^{(k)} \varphi^{(k)}, \tag{1.1}$$

For scalar or complex valued functions, the Hilbert space is $\mathcal{H} = L^2(\Theta)$, where a space vector $\boldsymbol{x}$ in the domain Θ is considered, and the time variable t, possessing an inner product defined by $(f, g) = \int_\Theta f_i g_i^* \, d\boldsymbol{x}$, with $*$-superscript denoting the conjugate transpose. The summation would allow for variable separation as

$$u(\boldsymbol{x}, t) = \sum_{k=1}^{\infty} v^{(k)}(t)\varphi^{(k)}(\boldsymbol{x}), \tag{1.2}$$

Considering the mean $\langle\cdot\rangle$, thought as "an average over a number of separate experiments", e.g. in the case of a function f with N_r realizations f_i, $\langle f \rangle = 1/N_r \sum_i f_i$, Holmes, Lumley & Berkooz (1996), the absolute value $|\cdot|$, and the 2-norm $||\cdot||$ defined as $||f||= (f, f)^{1/2}$, each normalized optimal basis φ is sought after

$$\max_{\varphi \in \mathcal{H}} \frac{\left\langle |(u, \varphi)|^2 \right\rangle}{||\varphi||^2}, \tag{1.3}$$

and through a condition on variations calculus, it can be shown equivalent to solving the eigenvalues ξ problem

$$\int_\Theta \langle u(\boldsymbol{x}, t) u^*(\boldsymbol{x}', t) \rangle \, \varphi(\boldsymbol{x}') \, d\boldsymbol{x}' = \xi \varphi(\boldsymbol{x}). \tag{1.4}$$

Moving from these continuous expressions to a *low-rank* approximation involves most of the time the Singular Value Decomposition (SVD) algorithm introduced in Burkardt, Gunzburger & Lee (2006). One can note that this method shares many similarities with the statistical technique of Principal Component Analysis, introduced in Pearson (1901), and recently reviewed in Jollife & Cadima (2016). With this algorithm, one can make a practical approximation by truncating the sum in (1.2) to a finite length L, first shown in Sirovich (1987), and expressed as

$$u^{\mathrm{POD}}(\boldsymbol{x}, t) = \sum_{k=1}^{L} v^{(k)}(t)\varphi^{(k)}(\boldsymbol{x}). \tag{1.5}$$

Subsequently, the *online* stage involves recovering the *expansion coefficients*, projecting back into our uncompressed, real-life space. This is where the separation between intrusive and non-intrusive methods appear, where the first is using techniques depending on the problem's

formulation, while the latter tries to statistically infer the mapping by considering the snapshots as a dataset.

1.2 Intrusive reduced-order methods: the Galerkin procedure

As described and improved in Couplet, Basdevant & Sagaut (2005), the conventional way to handle the second part of the Proper Orthogonal Decomposition approach to reduced-order modeling is the Galerkin procedure.

Let's consider a partial differential equation (PDE), defined by the nonlinear operator $\mathcal{N}$, with the x and t subscripts representing corresponding derivatives, as

$$u_t = \mathcal{N}_x u. \tag{1.6}$$

From the L-truncated sum in (1.5), each expansion coefficient $v^{(k)}$ is to be determined by the Galerkin procedure. By indeed reinjecting the approximated u^{POD} inside (1.6), and multiplying by the L POD modes $\boldsymbol{\varphi}$, known as Galerkin projection, a system of solvable equations is derived as

$$v_t^{(p)} = \sum_{i=1}^{L} \varphi^{(p)} \mathcal{N}_x u^{\mathrm{POD}} \approx \mathcal{R}^{(p)} u^{\mathrm{POD}}, \tag{1.7}$$

for the p-th expansion coefficient, with $\mathcal{R}$ the nonlinear residuals.

This POD-Galerkin approach has been applied in subsequent work to Shallow Water equations problems like a dam break and flood predictions in Zokagoa & Soulaimani (2012a,1). However, as mentioned in these references and many others, if $\mathcal{R}$ is a general nonlinear operator, it is not apparent how to gain any speedup in the offline stage, i.e., solving 1.7, unless some approximations are made on $\mathcal{R}$. Furthermore, for parameter-dependent problems where multi-simulations are required, the reduced basis is parameter-dependent as well, and it is the case for uncertainty quantification problems. The usage of many RB may be necessary and finding a way to combine these bases to find an accurate solution isn't straightforward, as discussed in Amsallem & Farhat (2014); Hesthaven & Ubbiali (2018); Zokagoa & Soulaimani (2018).

1.3 Non-intrusive reduced-order methods: Polynomial Chaos Expansion

If one wants to make sense of this snapshot dataset and build a surrogate model able to recover the projection coefficients correctly, a modeling procedure has to be performed. While usual and straightforward techniques like polynomial interpolation seem appealing for this task, they're having trouble yielding usable results in the case of a low amount of samples, as pointed out in Barthelmann, Novak & Ritter (2000).

A different take has been explored within the Polynomial Chaos Expansion (PCE) realm, proposed in Ghanem & Spanos (1991). Using Hermite polynomials, and more precisely, a set of multivariate orthonormal polynomials Φ, Wiener's Chaos theory allows for modeling of the outputs as a stochastic process. Considering the previous expansion coefficients $v^{(k)}(t)$ as a stochastic process of the variable t, the PCE is defined as

$$v^{(k)}(t) = \sum_{\alpha \in C^L} c_\alpha^{(k)} \Phi_\alpha(t), \tag{1.8}$$

with α identifying polynomials following the right criteria in a set C^L, Sun, Pan & Choi (2019). However, stability issues may arise, and a new different approach using the B-Splines Bézier Elements based Method (BSBEM) aimed at addressing this has been developed in Abdedou & Soulaimani (2018). While it has shown excellent results, this approach can also suffer from the *curse of dimensionality*, a term coined half a century ago, Bellman (1966), that still has significant repercussions nowadays, as shown in Verleysen & François (2005). In simple words, it implies that many well-intentioned approaches perform well on small domains, but suffer from unpredicted and impractical problems when scaled up to broader contexts.

1.4 Modern use of Machine Learning for physical problems

1.4.1 Data-driven methods in Computational Fluid Dynamics

While Neural Networks have been around for a while now, traced back to the *perceptron* model, Rosenblatt (1958), they had to wait for the concept of *backpropagation* and *automatic differentiation*, coined respectively in Linnainmaa (1976) and Rumelhart, Hinton & Williams (1986), to have a computationally practical way of training their multi-layer, less trivial counterparts. Sub-

sequently, various other types of networks became popular, such as Recurrent Neural Networks, Hopfield (1982), and later Long-Short-Term Memory networks, Hochreiter & Schmidhuber (1997), that allowed for breakthroughs in sequenced data. While the universal approximation power of Deep Neural Networks, in the context of Deep Learning, had been predicted for a long time, especially in Rina Dechter (1986), the community had to wait till the early 2010s to finally have both the computational power and the practical tools to train these large networks, with the likes of Goodfellow, Bengio & Courville (2016); Hinton (2007), and it quickly led to breakthroughs, making sense of and building upon massive amounts of data, with work from Szegedy *et al.* (2017), Mikolov *et al.* (2013), and Karras *et al.* (2019) to only name a few.

Conventionally, laws of physics are expressed as well-defined PDEs, with boundary/initial conditions as constraints, but lately, pure data-driven methods gave birth to new approaches in PDE discovery, Brunton, Proctor & Kutz (2016). The take-off of this new field of Deep Learning in Computational Fluid Dynamics was predicted in Kutz (2017). Its flexibility allows for multiple applications, such as the recovery of missing CFD data as in Carlberg, Jameson, Kochenderfer, Morton, Peng & Witherden (2019), or aerodynamic design optimization, Tao & Sun (2019). The cost associated with a fine mesh is high and yet has been overcome with a Machine Learning approach aiming at assessing errors and correcting quantities in a more coarse setting, Hanna, Dinh, Youngblood & Bolotnov (2020). New research in the field of numerical schemes has been performed in Després & Jourdren (2020), presenting the Volume of Fluid-Machine Learning (VOF-ML) approach, applied in bi-material settings. A review of the vast landscape of possibilities is explored in Brunton & Kutz (2019). Additionally, an older study of available Machine Learning methods, yet applied to environmental sciences and specifically hydrology, was performed in Hsieh (2009).

Nonetheless, it is common in engineering to only have *sparse and noisy data* at our disposal, but intuitions or expert knowledge about the underlying physics. It led researchers to think about how to balance the need for data in these techniques with expert knowledge on the system such as governing equations, first detailed in Raissi, Perdikaris & Karniadakis (2017a), then extended to Neural Networks in Raissi, Perdikaris & Karniadakis (2019a) with applications on Computational

Fluid Dynamics, as well as in vibrations Raissi, Wang, Triantafyllou & Karniadakis (2019b). A few of these approaches will be detailed in Section 1.4.3.2.

1.4.2 Statistical Machine Learning: a simple curve-fitting example

Before going further in this review, we propose a study of the simple case of a one-dimensional curve-fitting example. This section is built upon Bishop (2006), and its outstanding introduction chapter.

1.4.2.1 A toy problem: the cubic function

Let's introduce a toy problem that we will reuse throughout this work to quickly assess the performance of our methods, borrowed from Lakshminarayanan *et al.* (2017). Over the interval $\Omega = [-4, 4]$, we sample $N = 20$ data points of the true function, $u(x) = x^3$, polluted by a Gaussian noise of mean 0 and standard deviation $\sigma_{\text{al}} = 9$. Each point (x_i, y_i) lives in the set $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$.

The real cubic function is also directly sampled from its true value $N_{\text{tst}} = 300$ times on a broader domain $\Omega_{\text{tst}} = [-6, 6]$. The general aim is to find a function $\hat{u}$ that is best approximating u. A plot of the data is shown in Figure 1.1.

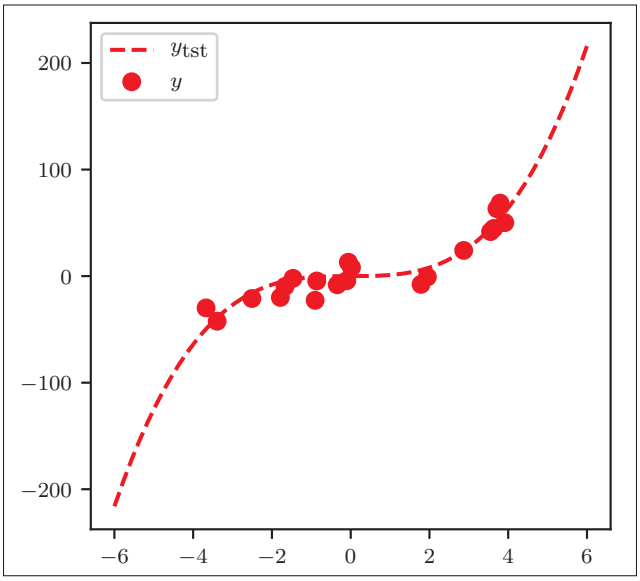


Figure 1.1 Toy problem: dataset and true value

1.4.2.2 Common polynomial fitting

As a first step, let's first try the simplest approach: fitting a polynomial function to our data, that we define as follows

$$\hat{u}(x, \boldsymbol{w}) = \sum_{i=0}^{d} w_i x^i, \tag{1.9}$$

with d the number of polynomial coefficients $\boldsymbol{w} = [w_0, \ldots, w_d]^\intercal$, which we will refer to as *weights* from now on since they define our model in the sense that changing them will change the prediction. To find the best possible $\hat{u}$, one needs to set up a *loss function*, also known as *cost function*, that, w.r.t. the weights, returns a metric assessing the quality of the predicted value of the model, and the most common simple choice is the Mean Squared Error, defined in our case as

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) := \frac{1}{N} \sum_{i=1}^{N} (\hat{u}(x_i; \boldsymbol{w}) - y_i)^2 \tag{1.10}$$

Minimizing this loss function w.r.t. the weights is analytically tractable by its quadratic nature, and an optimal solution exists, denoted by $\boldsymbol{w}^*$. In Figure 1.9, four different results are depicted, for four different values of d, the number of weights to be learned, i.e., the degree of the polynomial function, and one can clearly see the most simple definition of *overfitting* appearing in the $d = 10$ case: when the model is too complicated for the data and starts fitting the *noise* rather than the true hidden value. At the same time, one could say that $d = 1$ *underfits*, and that $d = 3$ seems like a good choice, no surprise here. A sample code is displayed in Figure 1.3, using NumPy's method `polyfit`, which is minimizing (1.10) under the hood.

Besides reducing the model complexity, overfitting can usually be overcome using more data, or *regularization*. While the former is trivial, the latter involves adding a penalty to the MSE loss, and the most common one is known as L2 regularization, or *weight decay* in Neural Networks contexts, Krogh & Hertz (1992). The regularized MSE is defined as, with $||\cdot||$ denoting the L2 norm,

$$\mathcal{L}^{\lambda}_{\text{MSE}}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w}) := \frac{1}{N} \sum_{i=1}^{N} (\hat{u}(x_i; \boldsymbol{w}) - y_i)^2 + \frac{\lambda}{2} ||\boldsymbol{w}||^2, \tag{1.11}$$

where λ is the *regularization coefficient*, and is a *hyperparameter* of the model, meaning that it is set externally.

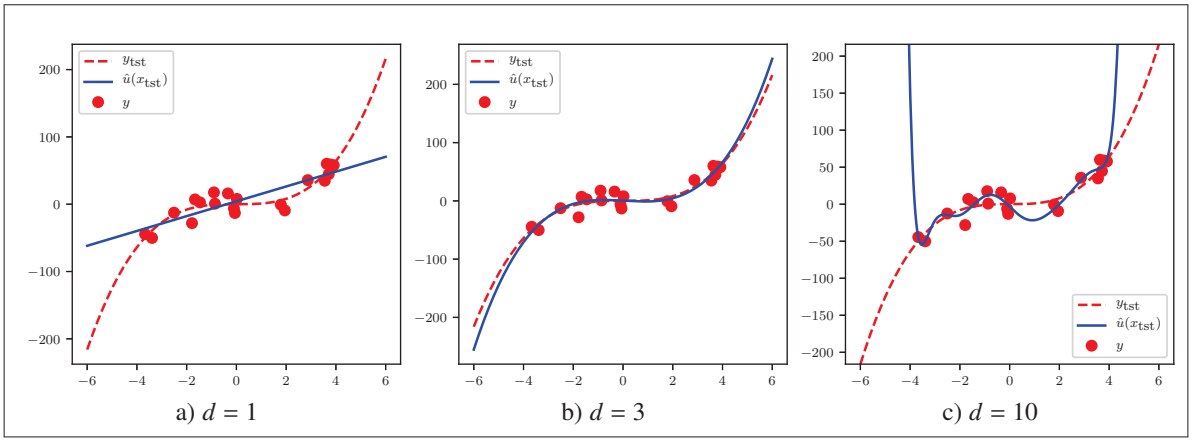


Figure 1.2 Toy problem: polynomial fitting, with three different degrees

```python
import numpy as np
d = 3
w_star = np.polyfit(x, y, deg=d)
print(f"Weights w* for d={d}: {w_star}")
P_star = np.poly1d(w_star)
y_pred = P_star(x_tst)
```

Figure 1.3 Sample code for the polynomial fitting of degree $d = 3$

Figure 1.4 shows the fight against overfitting for the overkill $d = 10$ model, with a L2-Regression on the left, and more training data on the right. Both approaches seem to help the predicted curve to be less prone to fit the noise.

While the examples are simple, the reasoning and the notions presented will be reused throughout this work.

1.4.2.3 The two sides of probabilities

The need for uncertainty accounting, as defined in Objective 3., forces us to keep a probabilistic view. In this section, we'll discuss *probability densities*, and leave discrete probabilities aside since we will only need continuous random variables in this work.

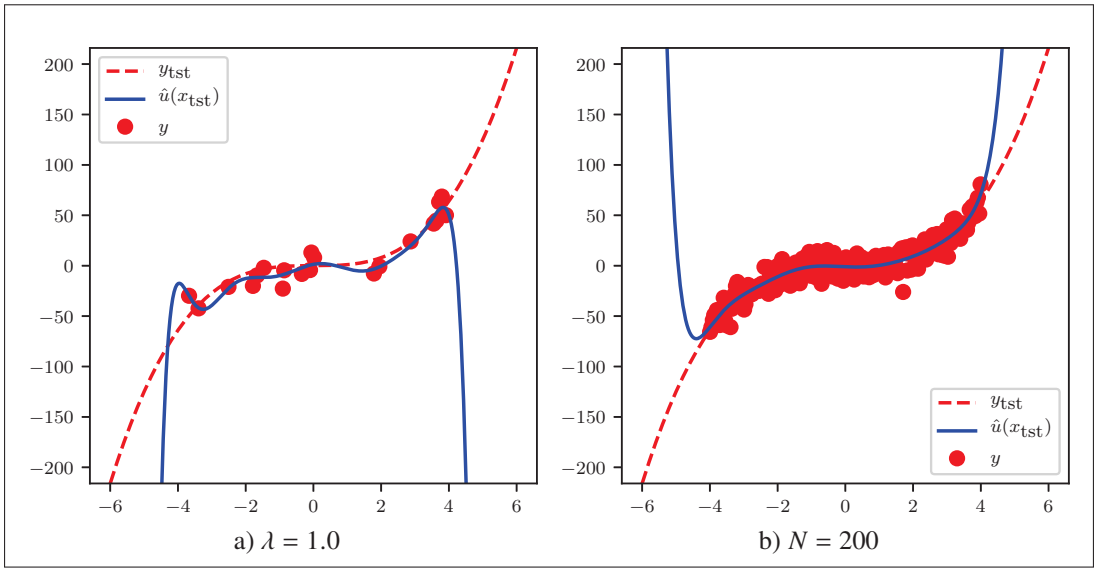


Figure 1.4 Toy problem: polynomial fitting, reducing overfitting from $d = 10$

As defined in Bishop (2006), $p(x)$ is a probability density of a real-valued variable x if $p(x \in (x, x + \delta x)) = p(x)\delta x$ for $\delta x \to 0$. In other words, it must respect the two conditions

$$p(x) \geq 0, \tag{1.12}$$

$$\int_{-\infty}^{\infty} p(x)\, dx = 1. \tag{1.13}$$

If n continuous variables are considered as $\boldsymbol{x} = [x_1, \ldots, x_n]^\intercal$, then the definition is analogous, and the same conditions apply—yet we integrate on the whole domain instead of just $\mathbb{R}$, and we refer to it as a *multivariate probability density*.

In both cases, the two main rules to keep in mind are the *sum rule* and the *product rule* and are respectively defined as

$$p(x) = \int p(x, y)\, dy \tag{1.14}$$

$$p(x, y) = p(y|x)p(x) \tag{1.15}$$

The most common associated quantities are the *expectation* of a function $f(x)$ under the probability distribution $p(x)$, defined as

$$\mathbb{E}[f] = \int p(x) f(x)\, dx, \tag{1.16}$$

and the *variance*, measuring the variability around the mean and writing as

$$\sigma^2[f] = \mathbb{E}\left[f - \mathbb{E}[f]^2\right]. \tag{1.17}$$

From the product rule, one can derive the famous Bayes' theorem: given two random variables x and y, the following relationship between conditional probabilities applies

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}, \tag{1.18}$$

and plays a key role in the second side of probabilities, known as the *bayesian* view, as opposed to the standard *frequentist* view. It changes from moving from seeing "probabilities in terms of the frequencies of random, repeatable events", to thinking about how "to convert a prior probability into a posterior probability by incorporation the evidence provided by the observed data", Bishop (2006).

If we indeed go back to our curve-fitting example of Section 1.4.2.2, we can write Bayes' theorem for our dataset $\mathcal{D}$, and the weights of our model $\boldsymbol{w}$

$$p(\boldsymbol{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}. \tag{1.19}$$

Each term above has a name, as follows:

1. $p(\boldsymbol{w})$: the *prior*, which is an assumption on the weights,
2. $p(\boldsymbol{w}|\mathcal{D})$: the *posterior*, given the dataset, what probabilities do multiple weights have,
3. $p(\mathcal{D}|\boldsymbol{w})$: the *likelihood*, given the weights, how likely it is to see the data,
4. $p(\mathcal{D})$: the *marginal likelihood*, which is a scaling term.

Since the last one is indeed not dependent on the weights $\boldsymbol{w}$, we can simply remember the following rule

$$\text{posterior} \propto \text{likelihood} \times \text{prior}. \tag{1.20}$$

A central element to the Bayesian view is the *predictive posterior distribution*, defined for a new pair (x, y) after seeing a dataset $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$ and accounting for all the weights configurations $\boldsymbol{w}$ as

$$p(y|x, \boldsymbol{x}, \boldsymbol{y}) = \int p(y|x, \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{x}, \boldsymbol{y})\, d\boldsymbol{w}. \tag{1.21}$$

1.4.2.4 Bayesian polynomial curve-fitting

We now assume that any target values y knowing the input x follows a normal distribution, with the mean equal to $\hat{u}(x; \boldsymbol{w})$. Still following Bishop (2006), we denote β the *precision*, which is the inverse of the variance, β^{-1}, and is itself a hyperparameter.
We, therefore, can write the probability of seeing the targets provided the inputs

$$p(y|x, \boldsymbol{w}, \beta) = \mathcal{N}(y|\hat{u}(x; \boldsymbol{w}), \beta^{-1}), \tag{1.22}$$

and we recall that a normal or Gaussian distribution writes as follows, for a mean μ and a variance σ^2

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right). \tag{1.23}$$

Still considering our dataset $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$, we know want to maximize the likelihood of seeing our targets, which writes as

$$p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \beta) = \prod_{i=1}^{N} \mathcal{N}(y_i|\hat{u}(x_i, \boldsymbol{w}), \beta^{-1}), \tag{1.24}$$

and that we can rewrite as a *log-likelihood*, injecting the normal definition

$$\ln p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \beta) = -\frac{\beta}{2} \sum_{i=1}^{N} (\hat{u}(x_i; \boldsymbol{w}) - y_i)^2 + \frac{N}{2} \ln \beta - \frac{N}{2} \ln 2\pi, \tag{1.25}$$

noting that the last two terms wouldn't have any impact on the maximization, and don't have to be accounted for when optimizing.

Equivalently, the opposite quantity can be minimized, and is known as the *Negative Log-Likelihood* (NLL). Its optimization w.r.t. the weights $\boldsymbol{w}$ gives the optimal $\boldsymbol{w}_*$, which maximizes the likelihood of seeing the targets knowing the parameters. Subsequently, minimizing w.r.t. β gives the Maximum Likelihood variance, which is an average across the domain, expressed as

$$\frac{1}{\beta_*} = \frac{1}{N}\sum_{i=1}^{N}\left[\hat{u}(x_i;\boldsymbol{w}_*) - y_i\right]^2 . \tag{1.26}$$

Let's now introduce a prior on the weights, $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{0}, \alpha^{-1}\boldsymbol{I})$, with α denoting the precision of this second distribution. Using Bayes' theorem, we finally get the Maximum A Posteriori (MAP) estimator, that is given by the minimum of the negative log form, Bishop (2006),

$$-\ln \text{MAP} = \frac{\beta}{2}\sum_{i=1}^{N}\left[\hat{u}(x_i;\boldsymbol{w}) - y_i\right]^2 + \frac{\alpha}{2}\boldsymbol{w}^\intercal\boldsymbol{w}, \tag{1.27}$$

which we recognize as being equivalent to a regularized Sum of Squared Error, a non-averaged version of the L2-regularized Mean Squared Error presented in (1.11), with the regularization coefficient $\lambda = \alpha/\beta$. We, therefore, see that while the concept of *prior* has been incorporated, this treatment still isn't fully Bayesian.

The requirement is to write the predictive posterior distribution for a new pair (x, y) as

$$p(y|x, \boldsymbol{x}, \boldsymbol{y}, \alpha, \beta) = \int p(y|x, \boldsymbol{w}, \beta)\, p(\boldsymbol{w}|\boldsymbol{x}, \boldsymbol{y}, \alpha, \beta)\, d\boldsymbol{w}. \tag{1.28}$$

This implies summing over all the possible weights, and is often the bottleneck for full Bayesian treatments. In this case it's analytically tractable, thanks to the Gaussian prior and the simple polynomial approach, which is linear w.r.t. to the polynomial basis. Nonetheless in most cases, it's intractable. The analytical case is given in Bishop (2006), and the predictive distribution writes as

$$p(y|x, \boldsymbol{x}, \boldsymbol{y}, \alpha, \beta) = \mathcal{N}\left(y|\mu_{\text{ana}}(x), \sigma^2_{\text{ana}}(x)\right), \tag{1.29}$$

with the mean and variance of the distribution expressed as

$$\mu_{\text{ana}}(x) = \Phi(x)^\intercal \boldsymbol{S} \sum_{i=1}^{N} \Phi(x_i) y_i, \tag{1.30}$$

$$\sigma^2_{\text{ana}}(x) = \beta^{-1} + \beta^{-1} \Phi(x)^\intercal \boldsymbol{S} \Phi(x). \tag{1.31}$$

In these results, we defined $\boldsymbol{S}^{-1} = \dfrac{\alpha}{\beta}\boldsymbol{I} + \sum_{i=1}^{N} \Phi(x_i)\Phi(x_i)^\intercal$,
and Φ the vector of polynomial basis, defined as $\Phi_i(x) = x^i$ for $0 \leq i \leq d$.
It's interesting to note that the variance expressed in (1.31) is composed of two terms: β^{-1}, a direct consequence of the noise within the data, which will be later referred to as *aleatoric uncertainty*, as well as $\beta^{-1}\Phi(x)^\intercal \boldsymbol{S}\Phi(x)$, which accounts for the uncertainty on the weights $\boldsymbol{w}$, and represents the additional information that is provided by the Bayesian treatment, denoting the different ways the model could fit the data.
As an illustration, we apply these results to the toy problem presented in Section 1.4.2.1. The Python implementation is straightforward since (1.29) gives explicit definitions and is depicted in Figure 1.6, while the results are in Figure 1.5. We used $\beta = 1/\sigma^2_{\text{al}} = 1/3^2$, the inverse of the known noise, and $\alpha = 0.11$. We can notice in Figure 1.5 that the $d = 10$ case doesn't overfit and is a direct result of the built-in regularization shown in (1.27). One can note that this choice of α and β would produce a regularization coefficient of $\lambda = 1.0$ in the previous frequentist approach, justifying our pick of α.

1.4.2.5 Neural Network regression

First introduced as the perceptron in Rosenblatt (1958), artificial Neural Networks now represent the epicenter of modern Machine Learning. The Universal Approximation Theorem shows that, in the case of *width-bounded ReLU networks*, as stated in Lu, Pu, Wang, Hu & Wang (2017):

Theorem 1. *For any Lebesgue-integrable function $u : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully-connected ReLU network with width $d_m \leq n + 4$, such that the function $\hat{u}$ represented by this network satisfies*

$$\int_{\mathbb{R}^n} |u(x) - \hat{u}(x)|\, dx < \epsilon. \tag{1.32}$$

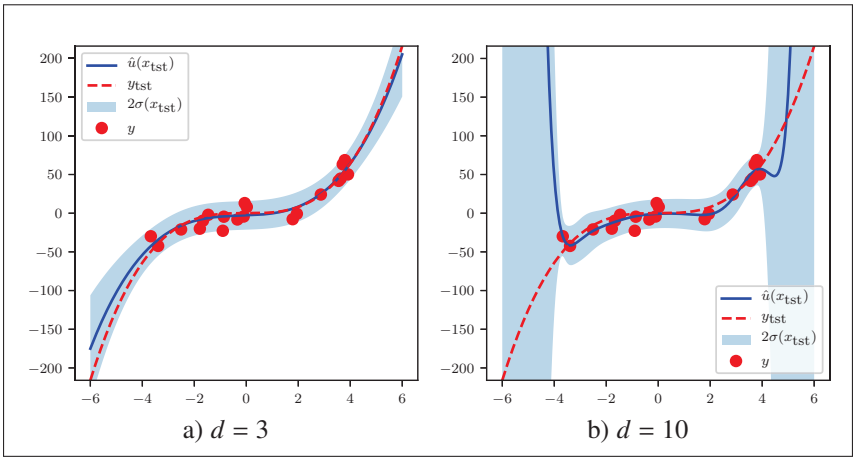


Figure 1.5 Toy problem. Polynomial Bayesian fitting

```
d = 10
beta = 1/noise_std**2
alpha = 0.11

# Polynomial basis
def phi(x_):
    return np.array([x_**i for i in range(d+1)])

# S matrix
S_inv = alpha/beta * np.identity(d+1) + \
        * np.array([phi(x_i).dot(phi(x_i).T) for x_i in x]) \
            .sum(0)
S = np.linalg.inv(S_inv)

# Mean for a given x
def mean(x_):
    sum_n = np.array([phi(x[i]) * y[i] for i in range(N)]) \
              .sum(0)
    return beta * phi(x_).T.dot(S).dot(sum_n)

# Variance for a given x
def variance(x_):
    return 1/beta + 1/beta*phi(x_).T.dot(S).dot(phi(x_))

# Predictions
y_pred = np.zeros_like(x_tst)
var = np.zeros_like(x_tst)
for i, x_i in enumerate(x_tst):
    y_pred[i] = mean(x_i)
    var[i] = variance(x_i)
sig = np.sqrt(var)
```

Figure 1.6 Python implementation of the full Bayesian Polynomial curve-fitting

Let's define the missing components. A *feedforward* Neural Network is a set of *neurons* organized in d *hidden layers*. Each hidden layer j has a *width* $l^{(j)}$. There's an additional *input*

layer, its width being tied to space in which the function we wish to approximated lives, $\mathbb{R}^n$ in the theorem for instance, and also an output layer, bringing the overall *depth* to be $d + 2$. To reach the result of Theorem 1, the authors of Lu *et al.* (2017) make the assumption of a depth of $4n + 1$. Figure 1.7 gives a representation of a network.

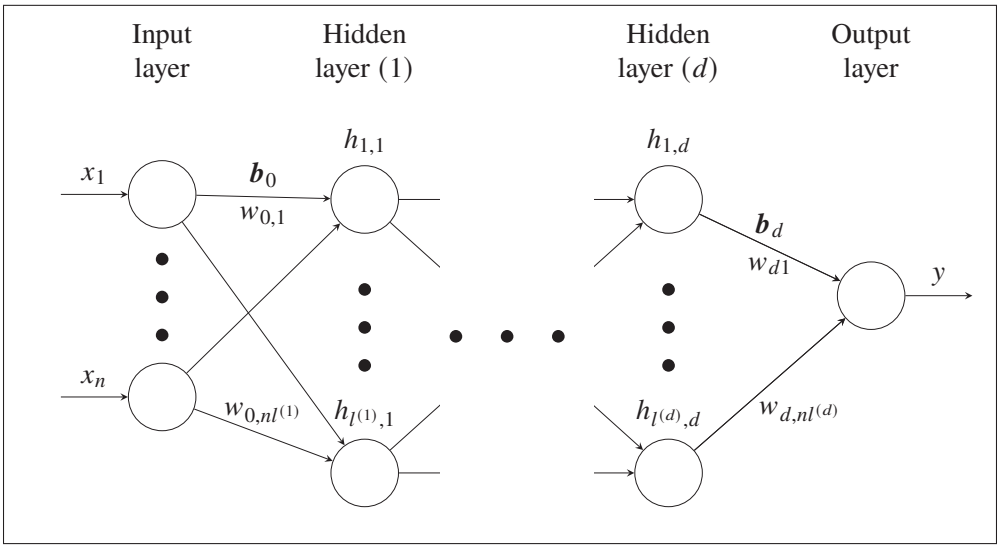


Figure 1.7 $\hat{u}(x_1, \ldots, x_n; \boldsymbol{w}, \boldsymbol{b}) = y$, a Deep Neural Network regression

Each layer j but the input has a *state* defined as a column vector $\boldsymbol{h}^{(j)}$, which is a linear combination of the precedent state (with $\boldsymbol{h}^{(0)} = [x_1, ..., x_n]^\intercal$ being the inputs) and the weights matrix $\boldsymbol{w}^{(j-1)}$ and bias column vector $\boldsymbol{b}^{(j-1)}$ of the layer, transformed by an *activation function* ϕ as following

$$\boldsymbol{h}^{(j)} = \phi\left(\boldsymbol{w}^{(j-1)}\boldsymbol{h}^{(j-1)} + \boldsymbol{b}^{(j-1)}\right). \tag{1.33}$$

Activation functions are nonlinearities, and the most common one is the Rectified Linear Unit, Nair & Hinton (2010), known as ReLU, and defined as

$$\phi_{\text{ReLU}}(x) = \max(0, x), \tag{1.34}$$

and are the reason why one is able to approximate most functions with these Neural Networks, or *multi-layer perceptrons*, as seen in Theorem 1. Yet, to make good predictions, the model *parameters*, i.e., the weights and the biases, need to have the right values.

Going back to our curve-fitting example, with our N-sized dataset $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$, one can measure the quality of the predictions $\hat{u}(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{b})$ through a loss function, and the MSE defined in (1.11) also applies here

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} [\hat{u}(\boldsymbol{x}_i; \boldsymbol{w}, \boldsymbol{b}) - \boldsymbol{y}_i]^2 + \lambda ||\boldsymbol{w}||^2, \tag{1.35}$$

to which we've also applied the same *weight decay*, L2 regularization.

Minimizing this loss function, i.e., training the network, can't, unfortunately, be performed analytically, and have to be handled by an *optimizer*. In the case of optimizers based on Stochastic Gradient Descent, such as Adam, Kingma & Ba (2014), we take the derivative of this loss $\mathcal{L}$ w.r.t. the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ in order to update them in a step similar to the following equation, using a concept called *backpropagation*, Linnainmaa (1976). The weights $\boldsymbol{w}^{n+1}$ and biases $\boldsymbol{b}^{n+1}$ corresponding to the epoch $n + 1$ write as

$$\left(\boldsymbol{w}^{n+1}, \boldsymbol{b}^{n+1}\right) = (\boldsymbol{w}^n, \boldsymbol{b}^n) - \eta f \left(\frac{\partial \mathcal{L}(\boldsymbol{w}^n, \boldsymbol{b}^n; \boldsymbol{x}, \boldsymbol{y})}{\partial (\boldsymbol{w}^n, \boldsymbol{b}^n)} \right), \tag{1.36}$$

with $f(\cdot)$ being a function of loss derivative w.r.t. the weights and biases that depends on the optimizer choice, and η the *learning rate*, a hyperparameter for SDG training.

In order to perform our NN regression on the cubic toy problem, we choose a network *topology* of $d = 3$ layers of $l^{(1)} = l^{(2)} = l^{(3)} = 5$ neurons, to follow exactly the minimum requirement of $n + 4$ in Theorem 1. The ReLU nonlinearity is used, as well as a learning rate of $\eta = 0.05$, and a regularization coefficient of $\lambda = 0.1$. Training is performed by the Adam optimizer, for $N_e = 5000$ epochs. A sample of Python code is presented in Figure 1.8, using the TensorFlow library, Abadi (2016), in version 2. The results are displayed in Figure 1.9.

1.4.3 Physics-informed Machine Learning

In this section, we will present a few papers from researchers at Brown University and the University of Pennsylvania, that paved the way for *physics-informed machine learning* and first caught our attention for an application in our field of small and expensive data.

```python
import tensorflow as tf
tfk = tf.keras

model = tfk.Sequential([
    *[tfk.layers.Dense(5,
        activation="relu",
        kernel_regularizer=tfk.regularizers.l2(0.1)) for _ in range(3)],
    tfk.layers.Dense(1),
])
model.compile(optimizer=tfk.optimizers.Adam(0.1), loss="mse")
model.fit(x, y, epochs=1000, verbose=0)
y_pred = model.predict(x_tst)
```

Figure 1.8 Toy Problem. TensorFlow 2 implementation of a NN regression.

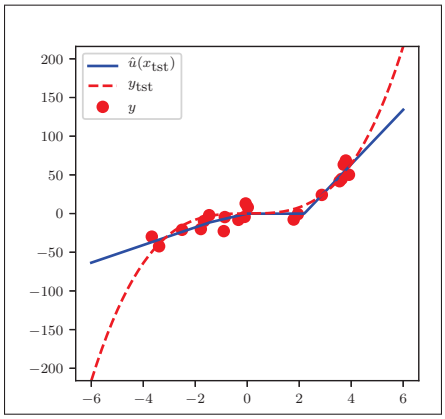


Figure 1.9 Toy problem. NN curve-fitting

Since this group of methods has been investigated a lot for this work and has remained an inspiration, we will dive in a bit more into details in this review.

1.4.3.1 Encoding physics in Gaussian Processes

"A Gaussian process is a generalization of the Gaussian probability distribution.", Rasmussen & Williams (2006). It has a *mean* (here 0) and a *covariance function* k, for instance, the Square Exponential (see 1.39). It could be thought of as a very long vector containing every

function value $y_i = f(x_i)$ defined as, with f' representing the test outputs of x', not yet observed,

$$f(x) \sim \mathcal{GP}(0, k(x, x'; \theta)) \tag{1.37}$$

$$\equiv \begin{bmatrix} f \\ f' \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} k(x,x;\theta) & k(x,x';\theta) \\ k(x',x;\theta) & k(x',x';\theta) \end{bmatrix}\right), \tag{1.38}$$

and the covariance can be for instance Gaussian, written as

$$k\left(x, x'; \begin{bmatrix} \alpha \\ \beta \end{bmatrix}\right) := \alpha^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{n} \frac{(x_d - x'_d)^2}{\beta_d^2}\right). \tag{1.39}$$

It would be used if one wanted to set the *prior* information of our functions to be smooth. Three random picks could be the ones displayed on the top of Figure 1.10. After taking into account five data points (in blue), we retrieve a *posterior*, which leads to a prediction concerning the mean (in dashed red) and standard deviation (with twice this pictured as a light blue area). Three random picks in this posterior are also displayed.

Setup

Let's now consider a time-dependent linear PDE, as presented in Raissi, Perdikaris & Karniadakis (2017b)

$$u_t = \mathcal{L}_x u, \quad x \in \Omega, \quad t \in [0, T]. \tag{1.40}$$

Applying the simplest time discretization scheme, Forward Euler, gives

$$u^n = u^{n-1} + \Delta t \mathcal{L}_x u^{n-1}, \tag{1.41}$$

and then placing a GP prior

$$u^{n-1}(x) \sim \mathcal{GP}(0, k_{u,u}^{n-1,n-1}(x, x', \theta)), \tag{1.42}$$

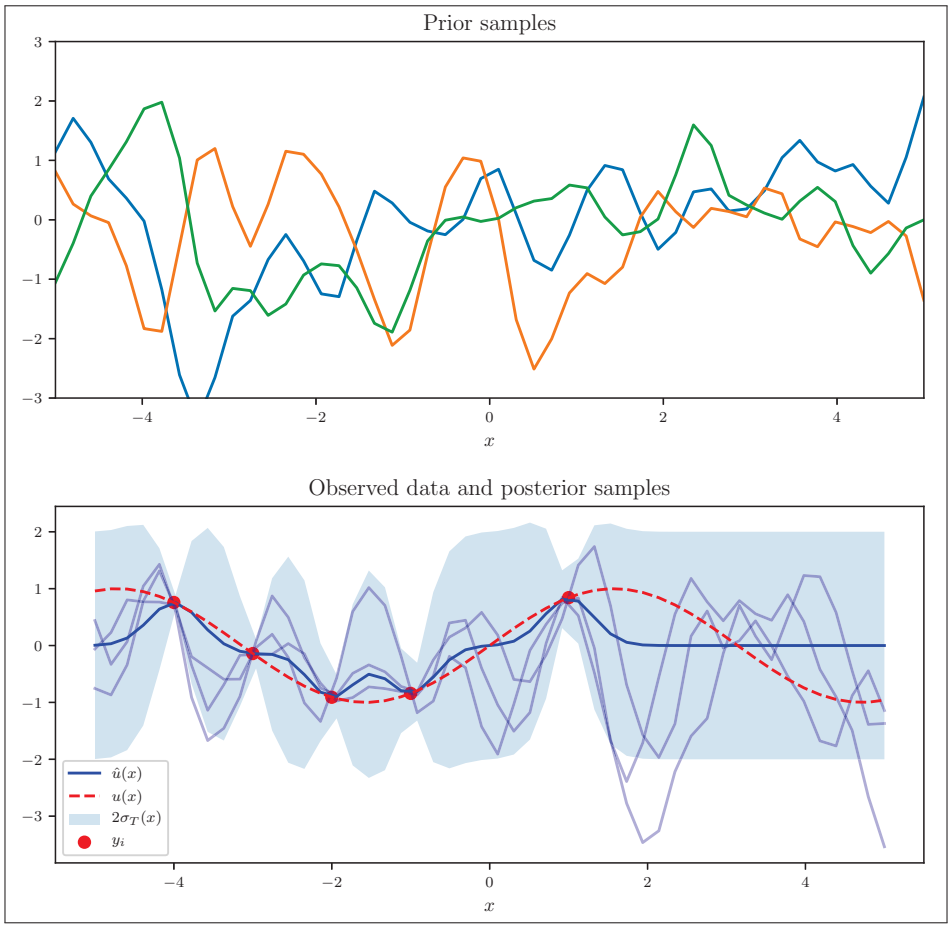


Figure 1.10 Gaussian Processes: samples generated using code from Bailey (2016)

therefore capturing the Euler rule in the following multi-output GP

$$\begin{bmatrix} u^n \\ u^{n-1} \end{bmatrix} \sim \mathcal{GP}\left(0, \begin{bmatrix} k_{u,u}^{n,n} & k_{u,u}^{n,n-1} \\ & k_{u,u}^{n-1,n-1} \end{bmatrix}\right). \tag{1.43}$$

Workflow

1. Train hyperparameters θ with initial $\{\boldsymbol{x}^0, \boldsymbol{u}^0\}$ and boundary $\{\boldsymbol{x}_b^1, \boldsymbol{u}_b^1\}$ data.

2. Predict *artificial data* $\{\boldsymbol{x}^1, \boldsymbol{u}^1\}$ of the next time-step from the posterior.

3. Train new hyperparameters for the time-step 2, using these *artificial data* $\{\boldsymbol{x}^1, \boldsymbol{u}^1\}$ and the boundary data $\{\boldsymbol{x}_b^2, \boldsymbol{u}_b^2\}$.

4. Predict new *artificial data* $\{\boldsymbol{x}^2, \boldsymbol{u}^2\}$ with these new hyperparameters.

5. Repeat 3. and 4. until the final time-step.

Handling nonlinearities

What if $\mathcal{L}_x$ is actually nonlinear? For example, Burgers' equation

$$u_t + uu_x = \nu u_{xx}, \quad \text{with} \quad \mathcal{L}_x := \nu u_{xx} - uu_x. \tag{1.44}$$

Applying Backward Euler gives

$$u^n = u^{n-1} - \Delta t \; u^n \frac{d}{dx} u^n + \nu \Delta t \frac{d^2}{dx^2} u^n. \tag{1.45}$$

Here, assuming u^n as a GP directly won't work since the nonlinear term $u^n \frac{d}{dx} u^n$ won't result in a GP. The trick here is to use the posterior mean of the previous step, denoted μ^{n-1}, giving a workable expression

$$u^n = u^{n-1} - \Delta t \; \mu^{n-1} \frac{d}{dx} u^n + \nu \Delta t \frac{d^2}{dx^2} u^n. \tag{1.46}$$

Limits on this approach include the cubic scaling with the number of training points of the computing power, due to the matrix inversion while predicting, and the case-by-case basis that is needed to treat nonlinear equations. This has lead researchers to look into tools that have the nonlinearities built-in: Deep Neural Networks.

1.4.3.2 Physics-Informed Neural Networks

As presented in Raissi *et al.* (2019a), let's consider a generic, parametrized nonlinear PDE

$$u_t + \mathcal{N}_x^\gamma u = 0, x \in \Omega, t \in [0, T]. \tag{1.47}$$

Whether we aim at solving it or identifying the parameters γ, the idea of the paper is the same: approximating $u(t, x)$ with a Deep Neural Network, and therefore defining the resulting

physics-informed neural network $f(t,x)$:

$$f := u_t + \mathcal{N}_x^\gamma u \tag{1.48}$$

This time, the trick is in the derivation of this special network, using automatic differentiation, Rumelhart *et al.* (1986), a chain-rule-based technique notoriously used in standard Deep Learning contexts, which in our case removes the need for numerical or symbolic differentiation.
As a test case, the authors consider Burgers' equation, expressed as followed in 1D with Dirichlet boundary conditions

$$\begin{gathered} u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1,1], \quad t \in [0,1], \\ u(0,x) = -\sin(\pi x), \\ u(t,-1) = u(t,1) = 0. \end{gathered} \tag{1.49}$$

From this, one can define $f(t,x)$, the PINN, as

$$f := u_t + uu_x - (0.01/\pi)u_{xx}. \tag{1.50}$$

The shared parameters are learned minimizing a custom version of the commonly used Mean Squared Error loss, with $\{t_u^i, x_u^i, u^i\}_{i=1}^{N_u}$ and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ respectively the initial/boundary data on $u(t,x)$ and collocations points for $f(t,x)$.

$$MSE = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2 + \frac{1}{N_f}\sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2 \tag{1.51}$$

The results of this benchmark problem can be found in Figure 1.11, that was generated using the original source code of Raissi *et al.* (2019a), that we ported to TensorFlow 2.0 and hosted on https://github.com/pierremtb/PINNs-TF2.0.
To show how easy it is to implement thanks to the modern open-source Deep Learning libraries, Algorithm 1.1 provides a pseudo-code example using TensorFlow and its automatic differentiation *gradients* function, Abadi (2016).

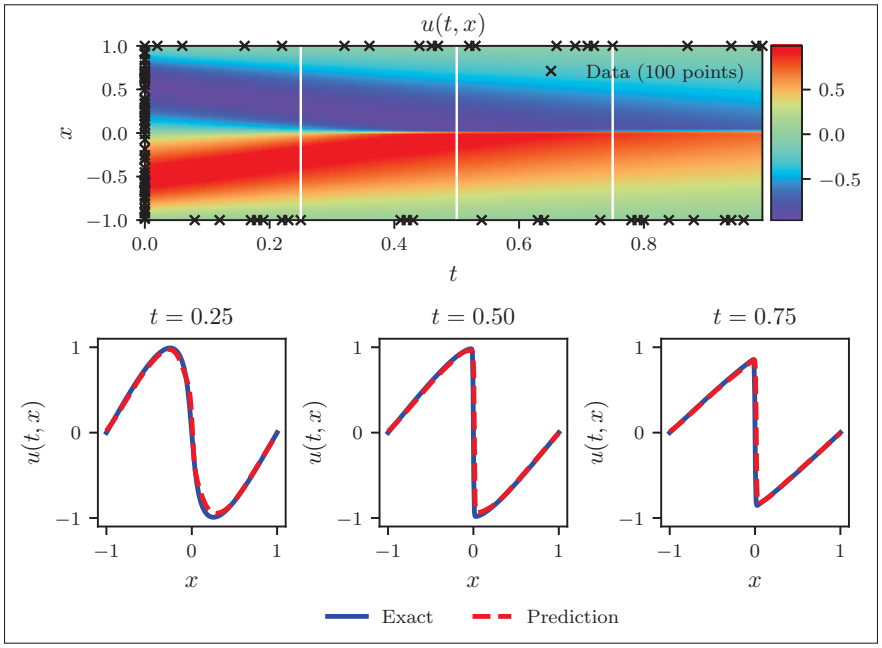


Figure 1.11 From top to bottom: predicted solution (with the initial and boundary training data), and a comparison predicted/exact solutions for the three snapshots (white vertical lines on top)

Algorithm 1.1 Implementing a PINN is straightforward with modern tools

Function `u` (t, x) **:**
  $\hat{u} \leftarrow neural\_net([x,t])$
return $\hat{u}$

Function `f` (t, x) **:**
  $\hat{u} \leftarrow u([x,t])$
  $\hat{u}_t \leftarrow tf.gradients(\hat{u}, t)$
  $\hat{u}_x \leftarrow tf.gradients(\hat{u}, x)$
  $\hat{u}_{xx} \leftarrow tf.gradients(\hat{u}_x, x)$
  $\hat{f} \leftarrow \hat{u}_t + \hat{u}\hat{u}_x - (0.01/\pi)\hat{u}_{xx}$
return $\hat{f}$

Further work has been performed by the same authors, applying the framework to different fields, including *Deep learning of vortex-induced vibrations*, Raissi *et al.* (2019b).

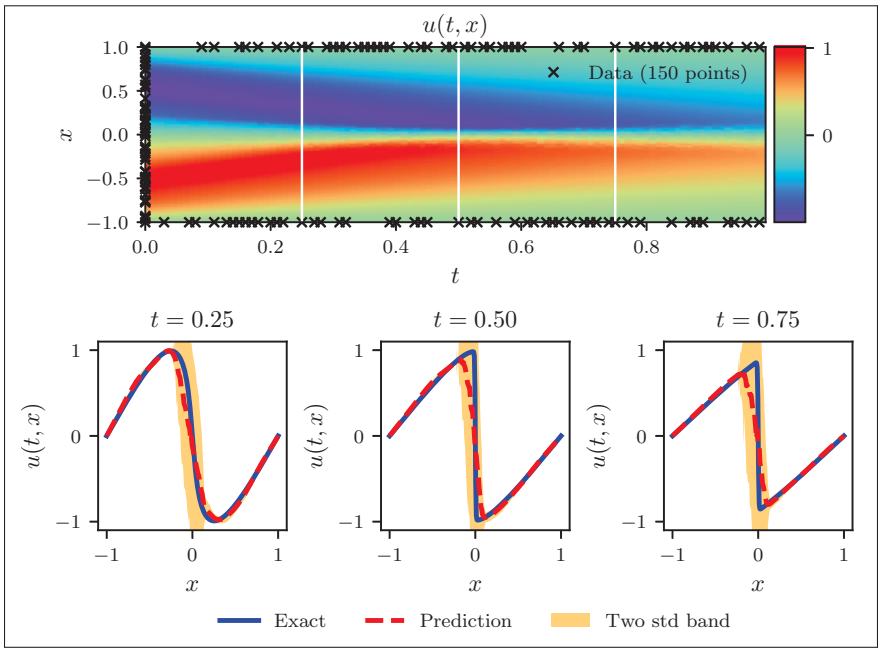


Figure 1.12 The same setup as in Figure 1.11, yet using the UQPINNs framework, and the data is polluted with uncorrelated noise of 10%.

1.4.3.3 Uncertainty Quantification in Physics-Informed Neural Networks

A quite recent addition to the mainstream Deep Learning catalog, *Generative Adversarial Networks*, proposed by Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville & Bengio (2014a), gained fame because of their ability to generate synthetic data. These networks are learning probability distributions through competition of a *generator*, creating fake data from random noise, and a *discriminator*, trying to pick the best out of an entry of the training set and the fake data at each iteration.

This technique has been used in Yang & Perdikaris (2019) to build an uncertainty quantifier within the Physics-Informed Neural Networks framework, which was one of the few missing pieces. In Figure 1.12, we can see the same benchmark as performed before, but this time it is featuring a confidence interval around the prediction. It was generated using the source code of Yang & Perdikaris (2019), that we ported to TensorFlow 2.0 and hosted on https://github.com/pierremtb/UQPINNs-TF2.0.

1.4.3.4 Hybridizing: Neural-Net-induced Gaussian Processes

Combining high expressivity of Deep NNs with the predictive ability and uncertainty quantification of GPs for regression is a concept named NNGP, as coined by the original authors Lee, Bahri, Novak, Schoenholz, Pennington & Sohl-Dickstein (2017). It achieved breakthrough results in image classification contexts, but has mostly not been used for other tasks. While it seems to be a promising approach, the main paper of interest that is applying it to our field, Pang, Yang & Karniadakis (2019), didn't provide any source code.

1.4.3.5 Discussion and benchmark

Table 1.1 Features comparison of the different physics-informed ML approaches

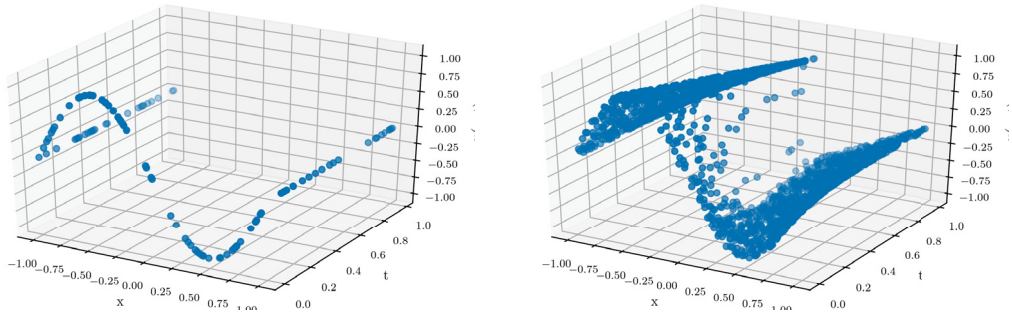
	GPs	*PINNs*	*UQPINNs*	*NNGPs*
UQ	Yes	No	Yes	Yes
Nonlinear	With tricks	Yes	Yes	With tricks

The aforementioned researchers have been very busy building up and extending various frameworks to tackle this new field of small-data, physics-informed machine learning. Table 1.1 compares the four main frameworks on their ability to propagate uncertainties and the handling of nonlinearities.

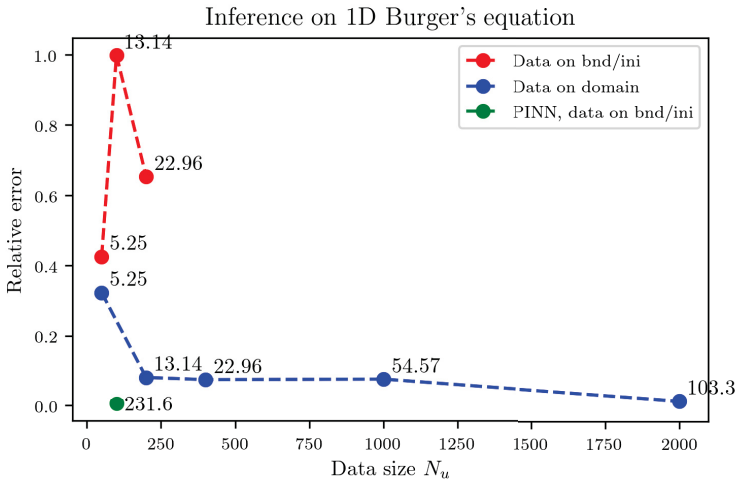
While the NNGP framework seems the most versatile, the PINN one seems like an excellent choice in the present context of easy use of Deep Learning libraries such as TensorFlow, Abadi (2016), and since the source code is provided by the authors.

Even if it's conceptually easy to understand and be confident in the fact that encoding the laws of physics in Neural Networks helps in predicting accurately, this section also aims at picturing it with a small experiment. For our system, we will provide a regular, physics-uninformed ML approach as close as possible to the physics-informed one.

The PINN for solving the 1D Burgers' equation has a topology of 8 hidden layers, each involving 20 neurons each. A set of initial and boundary data is fed to it, and an example of 100 is depicted on the left of Figure 1.13a.



a) Training data in different modes: 100 initial/boundary points only on the left, and 1000 domain-wide on the right



b) Performance results, with the relative error plotted against the data size used for training. The training time in seconds is used as label of each point.

Figure 1.13 Benchmarking the PINN approach

To make a physics-uninformed alternative, we use a regular Deep Neural Network with the same topology and feed it first with the same initial/boundary data, and then with domain-wide data to improve its accuracy and better compete, as shown on the right of Figure 1.13a. Benchmarking results are displayed in Figure 1.13b, and it confirms that the physical constraints of the PINN enable it to predict accurately with much less data, by a factor of 20 on this example. In green, one can observe the reference, i.e., the PINN result. With the dataset size horizontally and the relative error vertically, the optimum lies at the lower-left corner. The red points are results from feeding a physics-uninformed, regular DNN with boundary/initial data and is obviously

not performing well. Therefore, we tried feeding domain-wide data to the same uninformed network, and we see that the more data we feed, the better accuracy we get, and we tend to reach the same results as the PINN, yet with much more data involved (2000 v. 100).

1.5 Machine Learning-based non-intrusive reduced-order methods, and uncertainties

Mixing expert knowledge and data-driven approaches, the idea of modern *physics-informed* Machine Learning techniques provides many advantages. Yet despite the effectiveness that has just been demonstrated, the intrusive nature of the PINN framework—having to go back to the equation—doesn't make it an ideal solution for our use case, since it's contrary to our Objective 1..

After seeing the great results that modern Machine Learning techniques brought to intrusive methods in Section 1.4, it seems straightforward to also give them a try in a non-intrusive context. The use of Neural Networks has indeed been successfully applied to nonlinear problems, with examples on the Poisson and Navier-Stokes equations, in a novel framework coined as POD-NN in Hesthaven & Ubbiali (2018). It has subsequently been applied to time-dependent systems in Wang *et al.* (2019)—the primary building block of this work. The main idea is to directly infer the expansion coefficients v from (1.5) that have been of interest in this review with a multi-output Deep Neural Network $\hat{u}$, with its weights and biases denoted $\boldsymbol{w}$ for simplicity, such as

$$\boldsymbol{v}(t) = \hat{u}(t; \boldsymbol{w}). \tag{1.52}$$

This approach will be fully detailed and applied to a few test cases in Chapter 2.

While uncertainty quantification is still an open research area as far as Neural Networks are concerned, multiple attempts have been proposed to address it. As presented in Section 1.4.3.3, latent-variables in an adversarial context, like the UQPINNs from Yang & Perdikaris (2019) are one way. However, most of the work has been conducted embedding a Bayesian approach in Neural Networks, with the work of Mackay (1995), Barber & Bishop (1998), Graves (2011), Hernandez-Lobato & Adams (2015) ultimately leading to the backpropagation-compatible Bayesian Neural Networks defined in Blundell *et al.* (2015). Shortly after, a simpler and more practical take was presented with the Deep Ensembles framework in Lakshminarayanan *et al.*

(2017). And to the best of our knowledge, none of them have been applied within the POD-NN framework.

One should note that the POD-NN concept of Hesthaven & Ubbiali (2018) has also been revamped in Guo & Hesthaven (2018), using Gaussian Processes as a regression tool, instead of Deep Neural Networks, and in the case of time-dependent studies, LSTMs, Hochreiter & Schmidhuber (1997), have also been applied to perform this regression in Hu, Fang, Pain & Navon (2019).

Additionally, it is to be noted that a slightly different approach resides in Zhu & Zabaras (2018) and have also bundled uncertainty quantification, with the use of convolutional encoder-decoder Deep Neural Networks, with a few similarities in the compression-decompression approach of the UQPINNs framework.

CHAPTER 2

A DATA-DRIVEN NON-INTRUSIVE APPROACH: THE POD-NN FRAMEWORK

If the use of physics-informed constraints made a lot of sense and seamed appealing, their very intrusive nature wasn't a great fit for our problem, and there is a different way for AI tools such as Neural Networks to help in accelerating the modeling of physical phenomena. One example that has started to gain attention is the use of the regression power of Deep NNs to perform the final step in the process of creating Reduced-Order Models through Proper Orthogonal Decomposition, Hesthaven & Ubbiali (2018); Wang *et al.* (2019).

In this section, we will present the POD-NN framework, and apply it to three different benchmark problems: the 1D Shekel function, the 2D Ackley function, and an analytically-available solution of Burgers' equation, which is one-dimensional and time-dependent.

2.1 Problem Setup

In order to stay as general as possible, let's first mathematically define our objective.

We denote u the $\mathbb{R}^D$-valued function of interest. It depends on two types of parameters: the spatial ones $\boldsymbol{x} \in \mathbb{R}^n$, and the additional non-spatial ones $\boldsymbol{s} \in \mathbb{R}^P$—which could either be physical coefficients like fluid viscosity or time, such as

$$\begin{aligned} u : \ \mathbb{R}^{n+P} &\to \mathbb{R}^D \\ (\boldsymbol{x}, \boldsymbol{s}) &\mapsto u(\boldsymbol{x}, \boldsymbol{s}). \end{aligned} \tag{2.1}$$

Assuming computing this function costly, whether it be performing real-world measurements or running computationally-expensive simulations, one can only get a finite number of *snapshots* S, over a discretized space. We denote N_s the number of non-spatial parameters sampled, and N_t the number of time-steps considered—greater than one in a time-dependent setting, leading the total of snapshots to be $S = N_s N_t$. This space has n dimensions, with each direction $\vec{x}_i$, for simplicity, is for now discretized linearly in N_{x_i} nodes for $i \in (1, n)$. The total number of nodes N_D is hence defined as $N_D = \prod_{i=1}^{n} N_{x_i}$.

This spatial mesh being assumed fixed in time and known upfront, there is no reason to keep it as a parameter, and one can incorporate it in (2.1), removing $\boldsymbol{x}$ as a parameter in u, with $H = N_D \times D$ the total number of degrees of freedom (DOFs) on the mesh

$$\begin{aligned} u_D : \mathbb{R}^P &\to \mathbb{R}^H \\ \boldsymbol{s} &\mapsto u_D(\boldsymbol{s}). \end{aligned} \tag{2.2}$$

Sampling the S parameters this way results in a matrix of snapshots $\boldsymbol{U} = [u_D(\boldsymbol{s}_1)|\ldots|u_D(\boldsymbol{s}_S)] \in \mathbb{R}^{H\times S}$.

In this first section, we will go through the POD-NN approach of Hesthaven & Ubbiali (2018); Wang *et al.* (2019), and this will help us toward our first objectives: being fast to predict $u_D(\boldsymbol{s})$ for any new set of parameters $\boldsymbol{s}$, in a flexible way.

2.2 Reducing the order using Proper Orthogonal Decomposition

While this new regression objective u_D allows us to immediately take care of the space component, having a $\mathbb{R}^H$-valued output is not practical since its size can quickly become a problem—the size of a DNN with H as an output would be huge and close to impossible to train. This is when the Reduced-Order Model approach comes in.

Although there are other ways to reduce the order while keeping the time topology unchanged in a reduced-order model, such as Variational Autoencoders or Recurrent Neural Networks, the flexibility of the POD approach allows for the embedding of time as a regular parameter, Wang *et al.* (2019), and has been chosen in this work for its simplicity.

2.2.1 Compression via POD

Proper Orthogonal Decomposition aims at building a Reduced-Order Model (ROM) of a system, to produce a *low-rank approximation*, much more efficient to compute and use.

The theoretical goal is to find a function ψ in a Hilbert space that optimally represents $u(\boldsymbol{x}, \boldsymbol{s})$. This optimization problem is equivalent to an eigenvalues problem, as shown in Holmes *et al.*

(1997), from which we extract proper modes $\boldsymbol{\varphi}_i^D(\boldsymbol{x})$, such as

$$u_D(\boldsymbol{s}) = \sum_{i=1}^{\infty} \boldsymbol{v}_i(\boldsymbol{s})\boldsymbol{\varphi}_i^D, \tag{2.3}$$

with $\boldsymbol{v}_i$ being projection coefficients to determine. One can note the variable-separating nature of this approach since $\boldsymbol{\varphi}_i^D(\boldsymbol{x})$ encompasses the spatial mesh while $\boldsymbol{v}_i(\boldsymbol{s})$ handles the non-spatial parameters, which can include the time.

2.2.2 Finite truncation

Using the snapshots method from Sirovich (1987), one can efficiently extract a reduced POD basis in a finite-dimension context. One can make use of the Singular Value Decomposition algorithm, Burkardt *et al.* (2006), to extract $\boldsymbol{W} \in \mathbb{R}^{H\times H}$, $\boldsymbol{Z} \in \mathbb{R}^{S\times S}$, and the r descending-ordered positive singular values matrix $\boldsymbol{D} = \mathrm{diag}(\xi_1, \xi_2, \ldots, \xi_r)$ such as

$$\boldsymbol{U} = \boldsymbol{W} \begin{bmatrix} \boldsymbol{D} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{Z}^\top. \tag{2.4}$$

In order to get the first and most valuable L modes, one sets out a hyperparameter ϵ, to build up a truncating criterion defined as

$$\frac{\sum_{l=L+1}^{r} \xi_l^2}{\sum_{l=1}^{r} \xi_l^2} \leq \epsilon, \tag{2.5}$$

and build each reduced vector $\boldsymbol{V}_j \in \mathbb{R}^S$ from $\boldsymbol{U}$ and the j-th column of $\boldsymbol{Z}$, $\boldsymbol{Z}_j$, such as

$$\boldsymbol{V}_j = \frac{1}{\xi_j}\boldsymbol{U}\boldsymbol{Z}_j, \tag{2.6}$$

to form our POD basis

$$\boldsymbol{V} = \left[\boldsymbol{V}_1 | \ldots | \boldsymbol{V}_j | \ldots | \boldsymbol{V}_L\right] \in \mathbb{R}^{H\times L}. \tag{2.7}$$

Or equivalently, one can use $\boldsymbol{W}$ to build the same reduced basis.

2.2.3 Projection between spaces

Thanks to this reduction, one can now get projection coefficients corresponding to the snapshots

$$\boldsymbol{v} = \boldsymbol{V}^\intercal \boldsymbol{U}, \tag{2.8}$$

with $\boldsymbol{U}_{\text{POD}}$ the approximation of $\boldsymbol{U}$, projecting twice as

$$\boldsymbol{U}_{POD} = \boldsymbol{V}\boldsymbol{V}^\intercal \boldsymbol{U} = \boldsymbol{V}\boldsymbol{v}. \tag{2.9}$$

The relative projection error writes as

$$RE_{POD} = \sum_{j=1}^{S} \frac{||(\boldsymbol{U})_j - (\boldsymbol{U}_{POD})_j||_2}{||(\boldsymbol{U})_j||_2}, \tag{2.10}$$

with the $(\cdot)_j$ subscript denoting the j-th column of the targeted matrix, $||\cdot||_2$ the L2 norm, and the mean projection error over the samples and on each degree of freedom can be defined as

$$\boldsymbol{\sigma}_{POD} = \frac{1}{2S} \sum_{j=1}^{S} |(\boldsymbol{U})_j - (\boldsymbol{U}_{POD})_j| \in \mathbb{R}^H. \tag{2.11}$$

2.2.4 Improving POD speed for time-dependent problems

While the SVD algorithm is well-known and widely used, it can quickly get overwhelmed by the dimensionality of the problem, especially in a time-dependent context, such as Burgers' equation and their variations (Euler, Shallow Water, etc.), which will be discussed later in Section 2.4.3. Indeed, as time is being added as an input parameter, the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times S}$ can have a considerable width, making it very hard and long to process. One way to deal with this is the two-step POD algorithm, introduced in Wang *et al.* (2019).

Instead of invoking the algorithm directly on the wide matrix $\boldsymbol{U}$, the idea is to perform the SVD first on an unflattened tensor $\boldsymbol{U}_{3\text{D}} \in \mathbb{R}^{H \times N_S \times N_T}$, along the time axis for each parameter, as POD is usually used for standard space-time problems for a single parameter.

The workflow is as follows:

1. The "time-trajectory of each parameter value", quoting the authors' words, is being fed to the SVD algorithm, and the subsequent process of reconstructing a POD basis $\boldsymbol{T}_k$ is performed for each time-trajectory $\boldsymbol{U}_{3\mathrm{D}}^{(k)}$, with $k \in [1, N_S]$. A specific stopping hyperparameter, ϵ_0, is used here, producing L_0 reduced basis.

2. Each basis $\boldsymbol{T}_k$ is collected in a new time-trajectories matrix $\boldsymbol{T} \in \mathbb{R}^{H \times (N_S L_0)}$, on which the SVD algorithm is performed, with the regular ϵ hyperparameter forming the L reduced basis, and the final POD basis construction to produce $\boldsymbol{V}$ can happen.

A pseudo-code implementation is available in Algorithm 2.1, and a sample Python implementation using Numpy is shown in Figure 2.1.

Algorithm 2.1 Implementing the two-step POD that allows for large, time-dependent datasets handling

Function `POD` ($\boldsymbol{U}$, ϵ) **:**
    $\boldsymbol{D}, \boldsymbol{Z} \leftarrow SVD(\boldsymbol{U})$
    $\boldsymbol{\Lambda} \leftarrow \boldsymbol{D}^2$
    $\boldsymbol{\Lambda}_{trunc} \leftarrow \boldsymbol{\Lambda} \left[\frac{\sum_{i=0}^{L} \Lambda_i}{\sum_i \Lambda_i} \geq (1 - \epsilon) \right]$
    $\boldsymbol{V} \leftarrow \boldsymbol{U}.\boldsymbol{Z}.\boldsymbol{\Lambda}_{trunc}^{-1/2}$
return $\boldsymbol{V}$

Function `DualPOD` ($\boldsymbol{U}$, ϵ, ϵ_0) **:**
    $\boldsymbol{T} \leftarrow \boldsymbol{0}$
    for k *in* N_S **do**
        $\boldsymbol{T}_k \leftarrow \mathrm{POD}(\boldsymbol{U}^{(k)}, \epsilon_0)$
    end
    $\boldsymbol{V} \leftarrow \mathrm{POD}(\boldsymbol{T}, \epsilon)$
return $\boldsymbol{V}$

```python
import numpy as np
from numba import njit

@njit(parallel=False)
def perform_pod(U, eps):
    n_h, n_st = U.shape

    # SVD algorithm call, and reorienting
    _, D, ZT = np.linalg.svd(U, full_matrices=False)
    Z = ZT.T

    # Storing eigenvalues and their sum
    lambdas = D**2
    sum_lambdas = np.sum(lambdas)

    # Finding n_L and truncating
    n_L = 0
    sum_lambdas_trunc = 0.
    for i in range(n_st):
        sum_lambdas_trunc += lambdas[i]
        n_L += 1
        if sum_lambdas_trunc/sum_lambdas >= (1 - eps):
            break
    lambdas_trunc = lambdas[0:n_L]

    U = np.ascontiguousarray(U)
    V = np.zeros((n_h, n_L))
    for i in range(n_L):
        Z_i = np.ascontiguousarray(Z[:, i])
        V[:, i] = U.dot(Z_i) / np.sqrt(lambdas_trunc[i])

    return np.ascontiguousarray(V)

@njit(parallel=True)
def perform_dual_pod(U, eps, eps_init):
    n_h, n_s = U.shape

    # Init at the max it can be, n_t
    n_L_init = U.shape[1]

    T = np.zeros((n_h, n_L_init, n_s))
    for k in range(n_s):
        U_k = U[:, :, k]
        T_k = perform_pod(U_k, eps_init)
        n_L_k = T_k.shape[1]
        if n_L_k < n_L_init:
            n_L_init = n_L_k
        for i in range(n_L_init):
            T[:, i, k] = T_k[:, i]

    # Cropping the results accordingly and stacking
    T = np.ascontiguousarray(T[:, :n_L_init, :])

    # Reshaping the 3d-mat into a 2d-mat
    U_f = np.reshape(T, (n_h, n_s*n_L_init))
    return perform_pod(U_f, eps)
```

Figure 2.1 POD module. Sample Python 3 code, implementing POD algorithms

2.3 Learning the projection coefficients with Deep Neural Networks

2.3.1 Regression objective

For a ROM to be non-intrusive, computing the new solution for new non-spatial parameters $\boldsymbol{s}$ can be done online, using a *regression model*.

Since the spatial parameters $\boldsymbol{x}$ are getting reduced during the POD process, we can formalize the regression as a mapping u_{DB} that features the projection coefficients $\boldsymbol{v}(\boldsymbol{s})$ as outputs, such as

$$\begin{aligned} u_{DB} : \ \mathbb{R}^P &\rightarrow \mathbb{R}^L \\ \boldsymbol{s} &\mapsto \boldsymbol{v}(\boldsymbol{s}). \end{aligned} \tag{2.12}$$

2.3.2 Deep Neural Network and training

This mapping u_{DB} is then to be approximated by a Deep Neural Network $\hat{u}_{DB}(\boldsymbol{s}; \boldsymbol{w}, \boldsymbol{b})$, with $\boldsymbol{w}$ and $\boldsymbol{b}$ being the *weights* and *biases* of the network, that are learned during training (*offline* phase), and then reused during predictions (*online* phase). Its number of hidden layers is called the *depth* d—that is, without counting the input and output layers, and they each have a *width* $l^{(j)}$. A representation of this DNN is pictured in Figure 2.2, with d hidden layers—and therefore, $d + 2$ layers in total.

For a N-sized training dataset $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$, with the inputs $\boldsymbol{X}$ being the non-spatial parameters $\boldsymbol{s}$ after normalization and the projection coefficients $\boldsymbol{v}$ coming from a training/validation-split matrix of snapshots $\boldsymbol{U}$, a chosen optimizer runs a number of *training epochs* N_e in order to minimize the following Mean Squared Error loss function w.r.t. the network weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$:

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{b}; \boldsymbol{X}, \boldsymbol{v}) = \frac{1}{N} \sum_{i=1}^{N} [\hat{u}_{DB}(\boldsymbol{X}; \boldsymbol{w}, \boldsymbol{b})_i - (\boldsymbol{v})_i]^2 + \lambda ||\boldsymbol{w}||^2, \tag{2.13}$$

with the subscript i describing the i-th training point, λ the regularization parameter of the L2 regularization, as introduced in Section 1.4.2.5. The training phase is handled by the optimizer Adam, based on Stochastic Gradient Descent, also introduced in Section 1.4.2.5.

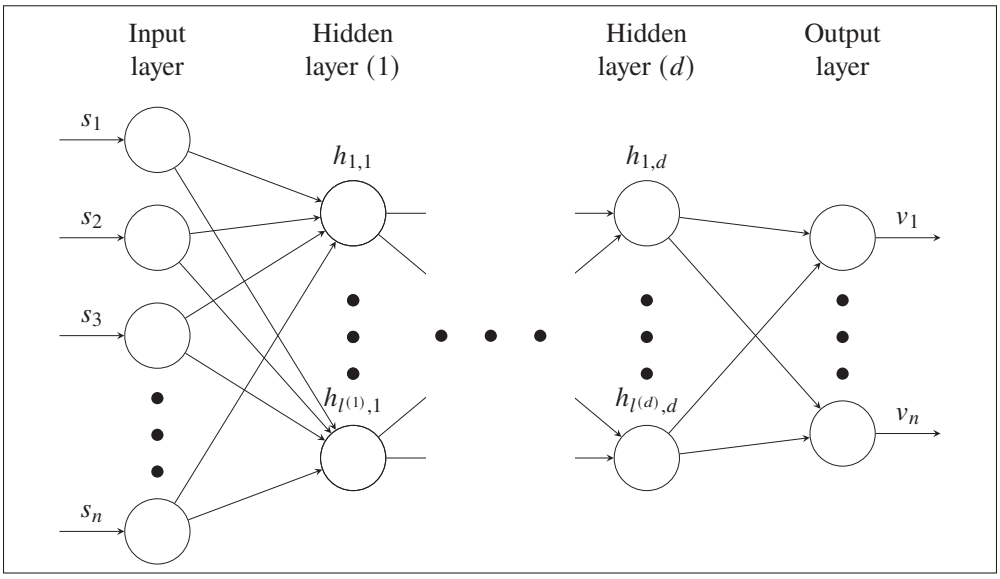


Figure 2.2 $\hat{\boldsymbol{v}} = \hat{u}_{DB}(\boldsymbol{X}; \boldsymbol{w}, \boldsymbol{b})$, a Deep Neural Network regression

2.3.3 Validation and testing metrics

Just as for the loss, we consider the widely-used *mean squared error* to assess the quality of the predictions, either on a N_{val}-sized validation dataset $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{v}_{\text{val}}\}$ or on a N_{tst}-sized validation dataset $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{v}_{\text{tst}}\}$.

$$MSE(\hat{\boldsymbol{v}}_{\text{val}}, \boldsymbol{v}_{\text{val}}, N_{\text{val}}) = \frac{1}{N_{\text{val}}} \sum_{i=1}^{N_{\text{val}}} \left[(\hat{\boldsymbol{v}}_{\text{val}})_i - (\boldsymbol{v}_{\text{val}})_i \right]^2, \tag{2.14}$$

with the inputs $\boldsymbol{X}_{\text{val}}$ being the non-spatial parameters $\boldsymbol{s}$ after normalization and the projection coefficients $\boldsymbol{v}_{\text{val}}$ coming from a validation-split matrix of snapshots $\boldsymbol{U}_{\text{val}}$, the *validation error* $E_{\text{val}} = MSE(\hat{\boldsymbol{v}}_{\text{val}}, \boldsymbol{v}_{\text{val}}; N_{\text{val}})$ is computed with $(\hat{\boldsymbol{v}}_{\text{val}})_i = \hat{u}_{DB}(\boldsymbol{X}_{\text{val}}; \boldsymbol{w}^k, \boldsymbol{b}^k)_i$, $\boldsymbol{w}^k$ and $\boldsymbol{b}^k$ being the weights and biases of the network at a training epoch k. The same goes for the *test error* E_T, which will be evaluated after the training ($k = N_e$).

Its relative counterpart RE carries more meaning to report the results:

$$RE(\hat{\boldsymbol{U}}, \boldsymbol{U}) = \frac{1}{S} \sum_{i=1}^{S} \frac{||\hat{\boldsymbol{U}}_i - \boldsymbol{U}_i||_2}{||\boldsymbol{U}_i||_2}, \tag{2.15}$$

with $\boldsymbol{U} = \boldsymbol{V}.\boldsymbol{v}$ and $\hat{\boldsymbol{U}} = \boldsymbol{V}.\hat{\boldsymbol{v}}$. In both cases $\boldsymbol{V}$ corresponds to the POD bases, extracted in Section 2.2.2.

During the training, we report three metrics, if possible: the training loss $\mathcal{L}$, the validation loss $\mathcal{L}_{\text{val}} = \mathcal{L}(\boldsymbol{w}, \boldsymbol{b}; \boldsymbol{X}_{\text{val}}, \boldsymbol{v}_T)$, as well as the validation relative error RE_{val}.

Sample codes for the custom *NeuralNetwork* class, with a custom training loop in TensorFlow 2, is available in Figure 2.4. Figure 2.3 also shows a sample implementation in Python of the *PODNNModel* wrapper class, which handles the model reduction and controls the *NeuralNetwork* object.

```python
import tensorflow as tf
import numpy as np

from .pod import perform_pod, perform_dual_pod
from .handling import pack_layers
from .logger import Logger
from .neuralnetwork import NeuralNetwork
from .metrics import re, re_s

class PodnnModel:
    # ...
    def initNN(self, h_layers, lr, lam):
        """Create the neural net model."""
        self.layers = pack_layers(self.n_d, h_layers, self.n_L)
        self.regnn = NeuralNetwork(self.layers, lr, lam,
                                   lb=self.lb, ub=self.ub)

    def train(self, X_v, v, X_v_val, v_val, epochs, freq=100):
        """Train the POD-NN's regression model, and save it."""
        # Validation and logging
        logger = Logger(epochs, freq)
        def get_val_err():
            v_val_pred = self.predict_v(X_v_val)
            return {
                "L_v": self.regnn.loss(v_val, v_val_pred),
                "RE_v": re_s(v_val.T, v_val_pred.T),
                }
        logger.set_val_err_fn(get_val_err)

        # Training
        self.regnn.fit(X_v, v, epochs, logger)

        # Saving
        self.save_model()
        return logger.get_logs()

    def predict(self, X_v):
        """Returns the predicted solutions, via proj coefficients."""
        v_pred = self.regnn.predict(X_v)
        U_pred = self.V.dot(v_pred.T)
        return U_pred

    @classmethod
    def load(cls, save_dir):
        """Recreate a pre-trained POD-NN model."""
        n_v, x_mesh, n_t = PodnnModel.load_setup_data(save_dir)
        podnnmodel = cls(save_dir, n_v, x_mesh, n_t)
        podnnmodel.load_train_data()
        podnnmodel.load_model()
        return podnnmodel
```

Figure 2.3 *PODNNModel* class: sample Python 3 code

```python
import tensorflow as tf

class NeuralNetwork:
    # ...
    def __init__(self, layers, lr, lam, model=None, lb=None, ub=None):
        self.model = tf.keras.Sequential()
        self.model.add(tf.keras.layers.InputLayer((layers[0],)))
        for width in layers[1:-1]:
            self.model.add(tf.keras.layers.Dense(width, tf.nn.tanh))
        self.model.add(tf.keras.layers.Dense(layers[-1], None))
        self.model.compile(optimizer=self.tf_optimizer, loss="mse")

    @tf.function
    def loss(self, v, v_hat):
        """Return a MSE loss function between the pred and val."""
        return tf.reduce_mean(tf.square(v - v_hat)) + self.regularization()

    @tf.function
    def grad(self, X, v):
        """Compute the loss and its derivatives w.r.t. the inputs."""
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(X)
            loss_value = self.loss(v, self.model(X))
            if self.adv_eps:
                # Adversarial contribution
                loss_x = tape.gradient(loss_value, X)
                X_adv = X + self.adv_eps * tf.math.sign(loss_x)
                loss_value += self.loss(v, self.model(X_adv))
        grads = tape.gradient(loss_value, self.wrap_training_variables())
        return loss_value, grads

    def regularization(self):
        l2_norms = [tf.nn.l2_loss(v) \
                        for v in self.wrap_training_variables()]
        l2_norm = tf.reduce_sum(l2_norms)
        return self.lam * l2_norm

    def fit(self, X_v, v, epochs, logger):
        """Train the model over a given dataset, and parameters."""
        X_v = self.normalize(X_v)
        v = self.tensor(v)

        # Optimizing
        for epoch in range(epochs):
            loss_value, grads = self.grad(X_v, v)
            self.tf_optimizer.apply_gradients(
                zip(grads, self.wrap_training_variables()))

    def predict(self, X):
        """Get the prediction for a new input X."""
        X = self.normalize(X)
        return self.model(X).numpy()
```

Figure 2.4 *NeuralNetwork* class: sample Python 3 code

2.4 Benchmark problems

In order to test the accuracy of the method, we'll present three benchmark problems from which analytical solutions can be generated.

The library TensorFlow, Abadi (2016), in version 2.1.0 is used for all results, and the SVD algorithm is performed by NumPy, all in Python 3.7. Documented source code will be made available at https://github.com/pierremtb/POD-UQNN, on the `POD-NN` branch.

Across the standard benchmarks, $\phi : x \mapsto \tanh(x)$ is used as *activation function* on all hidden layers, while a linear mapping is applied on the output layer since we need to retrieve real-valued variables. The chosen NN topology is $d = 2$ hidden layers, of widths $l^{(1)} = l^{(2)} = 64$. Normalizing is performed on all non-spatial parameters $\boldsymbol{s}$ to form the input $\boldsymbol{X}$ as

$$\boldsymbol{X} = \frac{\boldsymbol{s} - \bar{\boldsymbol{s}}}{\boldsymbol{s}_{\text{std}}}, \tag{2.16}$$

with $\bar{\boldsymbol{s}}$ and $\boldsymbol{s}_{\text{std}}$ being respectively the empirical mean and standard deviation over the dataset, on each column to keep physical meaning, e.g., the time would be normalized w.r.t. time moments.

Choice of optimizer

As it was used earlier in physics-informed machine learning research of Raissi *et al.* (2019a), the quasi-newton L-BGFS optimizer, Liu & Nocedal (1989), has been tried on the benchmark problems and proved to be overall less effective than Adam, Kingma & Ba (2014), as far as convergence speed and result accuracies were concerned. It is, however, to note that a combination of the two—starting with Adam for a few iterations and then finishing the optimization via L-BGFS, performed very well, yet required fine-tuning, minimizing the return on investment, so we chose not to use it in the following cases.

Datasets generation using native code

Simple benchmarks like the following are functions that we can call, and that output a result. One or more of their parameters is non-spatial and stochastic, i.e., to get a reference solution, we need to run it a rather large number of times.

Our baseline for these simulations is a sampling size of $N_{\text{val}} = 500$ for the training/validation sets, and N_{tst} for the test sets (which can be more prominent for time-dependent cases), leading to long computation times if it is done without optimization in Python. One way to drastically reduce it is through multi-threading and machine code Just-In-Time compilation, using the library Numba, Lam, Pitrou & Seibert (2015).

Two-step POD for time-dependent problems

Among the following benchmark, the only one that is time-dependent is the 1D Burgers' equation solution, Section 2.4.3. And that is where we make use of the two-step POD algorithm as described in Section 2.2.4. The results are comforting: on a dataset of size $S = 10,000$, with $N_t = 100$, the time to compute the SVD decomposition is reduced from 0.63 seconds to 0.51 when switching from the regular to the two-step POD process, which could result in a significant gain on more massive datasets. Numba optimizations have also been used for both the regular POD algorithm and dual one.

2.4.1 Shekel function (1D)

2.4.1.1 Definition

As a first benchmark, let's introduce the function presented in Shekel (1971), which is commonly used to test stochastic methods and is interesting in our case since it can take many stochastic parameters, which will be a great way to test if the POD-NN can perform well in this otherwise simple one-dimensional case. Being real-valued ($N_{\text{val}} = 1$) and one-dimensional in space, it is defined as

$$\begin{aligned} u : \ & \mathbb{R}^{1+P} \to \mathbb{R} \\ & (x, \boldsymbol{s}) \mapsto -\sum_{i=1}^{P/2} \left((x - s_{i+P/2})^2 + s_i \right)^{-1}, \end{aligned} \tag{2.17}$$

with a chosen central point for non-spatial parameters vector of size $P = 10$, borrowed from Abraham, Tsirikoglou, Miranda, Lacor, Contino & Ghorbaniasl (2018),

$$\boldsymbol{s} = \left[\frac{1}{10}(1, 2, 2, 4, 4), (4, 1, 8, 6, 3) \right]^{\mathsf{T}}.$$

2.4.1.2 Setup

The 1D space domain $\Omega_x = [0, 10]$ is discretized uniformly in $N_{x_1} = N_x = 300$, leading the number of DOFs to be $H = 300$ as well.

With $S = N_S = 500$ as our default number of samples of the parameters $\boldsymbol{s}$, and we use a Latin Hypercube Sampling (LHS) strategy to sample each non-spatial parameter s_i on a domain around the central point defined earlier in the function definition,

$$\Omega_{s_i} = \left[s_i \, (1 - \sqrt{3}/10), \; s_i \, (1 + \sqrt{3}/10) \right],$$

and generate the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times S}$.

After getting the reduced POD bases via (2.4–2.8) picking $\epsilon = 10^{-10}$, and generating a full set, the $(80\%, 20\%)$ *train/validation* ratio splits it into $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$ of size $N = 400$, $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{v}_{\text{val}}\}$ of size $N_{\text{val}} = 100$. We generate an additional test set $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{v}_{\text{tst}}\}$ of size $N_{\text{tst}} = 300$.

We choose a fixed learning rate of $\eta = 0.001$ for the Adam optimizer, as well as an L2 regularization with $\lambda = 10^{-4}$. No mini-batching is performed, i.e., the whole dataset is run through at once for each epoch. The training epochs number is set to $N_e = 15,000$.

2.4.1.3 Results

The initial dataset preparation was instantaneous, using the settings mentioned above on a standard desktop computer, while the training took 45 seconds on a regular desktop CPU. The relative errors reached were $RE_{\text{val}} = 4.06\%$ and $RE_{\text{tst}} = 4.03\%$ for validation and testing, respectively. In Figure 2.5 are displayed a few test predictions, as well as out-of-distribution predictions, sampled in the $\Omega_{s_{i,\text{out}}}$ defined as

$$\Omega_{s_{i,\text{out}}} = \left[s_i \, (1 - 1.5\sqrt{3}/10), \; s_i \, (1 - \sqrt{3}/10) \right] \cup \left[s_i \, (1 + \sqrt{3}/10), \; s_i \, (1 + 1.5\sqrt{3}/10) \right].$$

One can notice that while the predictions inside the dataset bounds (the three test samples in the first row) are excellent, performance really goes down on the second row, with predictions made *out-of-distribution*. A sample code of this benchmark lives in Figure 2.7.

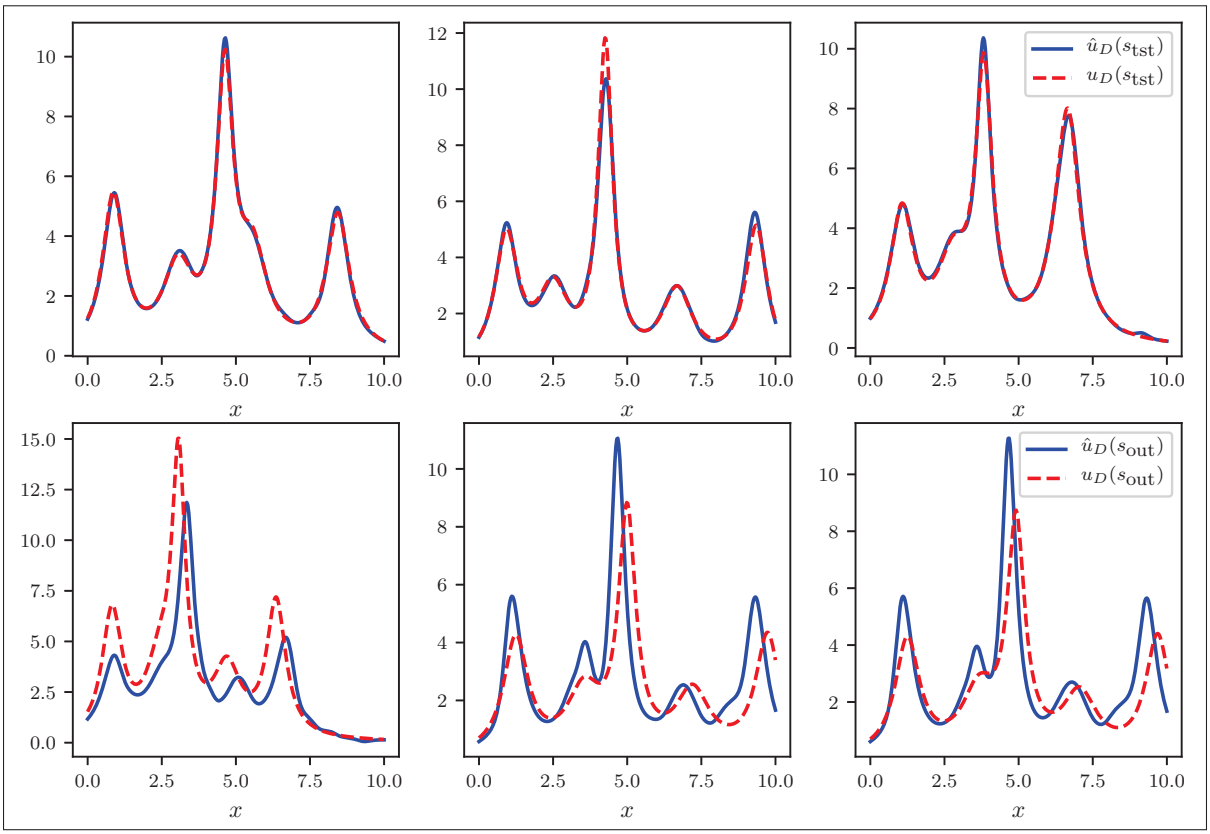


Figure 2.5 Shekel Function (1D). From left to right: comparing the predicted $\hat{u}_D$ and the observed data u_D from the dataset across three random snapshots of the test set. The second row shows samples s_{out}, outside the dataset bounds

2.4.1.4 Convergence study

To assess the convergence of the method, we put together a small systematic study on two hyperparameters that directly impact the results: the number of *training epochs* and the number of POD *snapshots*. Five different values for the snapshots count S are forming the x-axis of Figure 2.6, and three pairs of lines show the relative errors for different numbers of epochs N_e, with validation as dashed lines and testing as plain lines. As expected, one can see both parameters bring the relative errors down when they are increased, and the black pair shows that a low amount of epochs prevent the convergence with a larger snapshots count S.

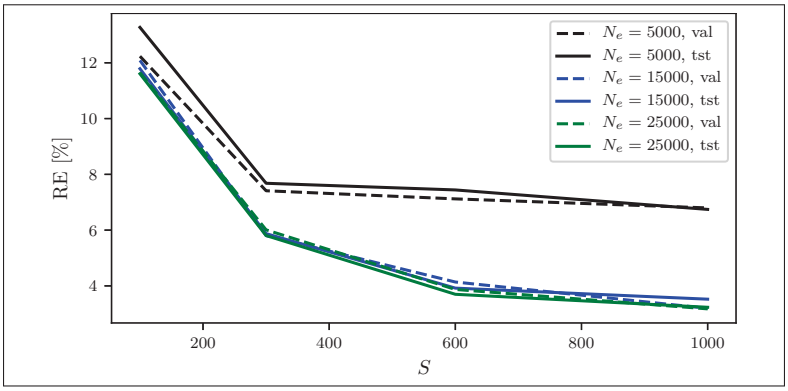


Figure 2.6 Shekel Function (1D). Systematic study results on the number of samples and training epochs.

```python
import numpy as np
from podnn.podnnmodel import PodnnModel
from podnn.metrics import re_s
from podnn.mesh import create_linear_mesh
from podnn.plotting import figsize, savefig
from hyperparams import HP as hp, u

#%% Prepare
x_mesh = create_linear_mesh(hp["x_min"], hp["x_max"], hp["n_x"])

#%% Init the model
model = PodnnModel("cache", hp["n_v"], x_mesh, hp["n_t"])

#%% Generate the dataset from the mesh and params
X_v_train, v_train, \
    X_v_val, v_val, \
    U_val = model.generate_dataset(u, hp["mu_min"], hp["mu_max"],
                                   hp["n_s"],
                                   hp["train_val"],
                                   hp["eps"])

#%% Train
model.initNN(hp["h_layers"], hp["lr"], hp["lambda"])
train_res = model.train(X_v_train, v_train, X_v_val, v_val, hp["epochs"],
                        hp["log_frequency"])

#%% Validation metrics
U_pred = model.predict(X_v_val)
err_val = re_s(U_val, U_pred)
print(f"RE_v: {err_val:4f}")

#%% Sample the new model to generate a test prediction
mu_lhs = model.sample_mu(hp["n_s_tst"],
                         np.array(hp["mu_min"]),
                         np.array(hp["mu_max"]))
X_v_tst, U_tst, _ = \
    model.create_snapshots(mu_lhs.shape[0], mu_lhs.shape[0],
                           model.n_d, model.n_h, u, mu_lhs)
U_pred = model.predict(X_v_tst)
print(f"RE_tst: {re_s(U_tst, U_pred):4f}")
```

Figure 2.7 Shekel Function (1D). Sample of Python 3 code to run the benchmark

2.4.2 Stochastic Ackley function (2D)

2.4.2.1 Definition

As a second benchmark, let's introduce a stochastic version of the Ackley function, producing a highly irregular surface with multiple extrema presented in Sun *et al.* (2019), which takes three parameters. Being real-valued ($D = 1$) and two-dimensional in space ($n = 2$), it is defined as

$$
\begin{aligned}
u : \mathbb{R}^{2+P} &\to \mathbb{R} \\
(x, y; \boldsymbol{s}) &\mapsto -20\,(1 + 0.1 s_3) \exp\left(-0.2(1 + 0.1 s_2)\sqrt{0.5(x^2 + y^2)}\right) \\
&\quad - \exp\left(0.5(\cos(2\pi(1 + 0.1 s_1)x) + \cos(2\pi(1 + 0.1 s_1)y))\right) \\
&\quad + 20 + \exp(0),
\end{aligned}
\tag{2.18}
$$

with the non-spatial parameters vector $\boldsymbol{s}$ of size $P = 3$, each element s_i uniformly sampled over $\Omega_{s_i} = [-1, 1]$, in the same way as the authors in Sun *et al.* (2019).

2.4.2.2 Setup

The 2D space domain $\Omega_{xy} = [-5, 5] \times [-5, 5]$ is discretized uniformly in $N_{x_1} = N_x = 400$ and $N_{x_2} = N_y = 400$, leading the number of DOFs to be $H = 160,000$.

With $S = 500$ as our default number of samples of the parameters $\boldsymbol{s}$, and we use an LHS strategy to sample each non-spatial parameter on their domain $[-1, 1]$ and generate the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times S}$.

After getting the reduced POD bases via (2.4–2.8) picking $\epsilon = 10^{-10}$, and generating a full set, the (80%, 20%) *train/validation* ratio splits it into $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{\nu}\}$ of size $N = 400$, $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{\nu}_{\text{val}}\}$ of size $N_{\text{val}} = 100$. We generate an additional test set $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{\nu}_{\text{tst}}\}$ of size $N_{\text{tst}} = 300$.

We choose a fixed learning rate of $\eta = 0.003$ for the Adam optimizer, as well as an L2 regularization with $\lambda = 10^{-4}$. No mini-batch split is performed. The training epochs number is set to $N_e = 50,000$.

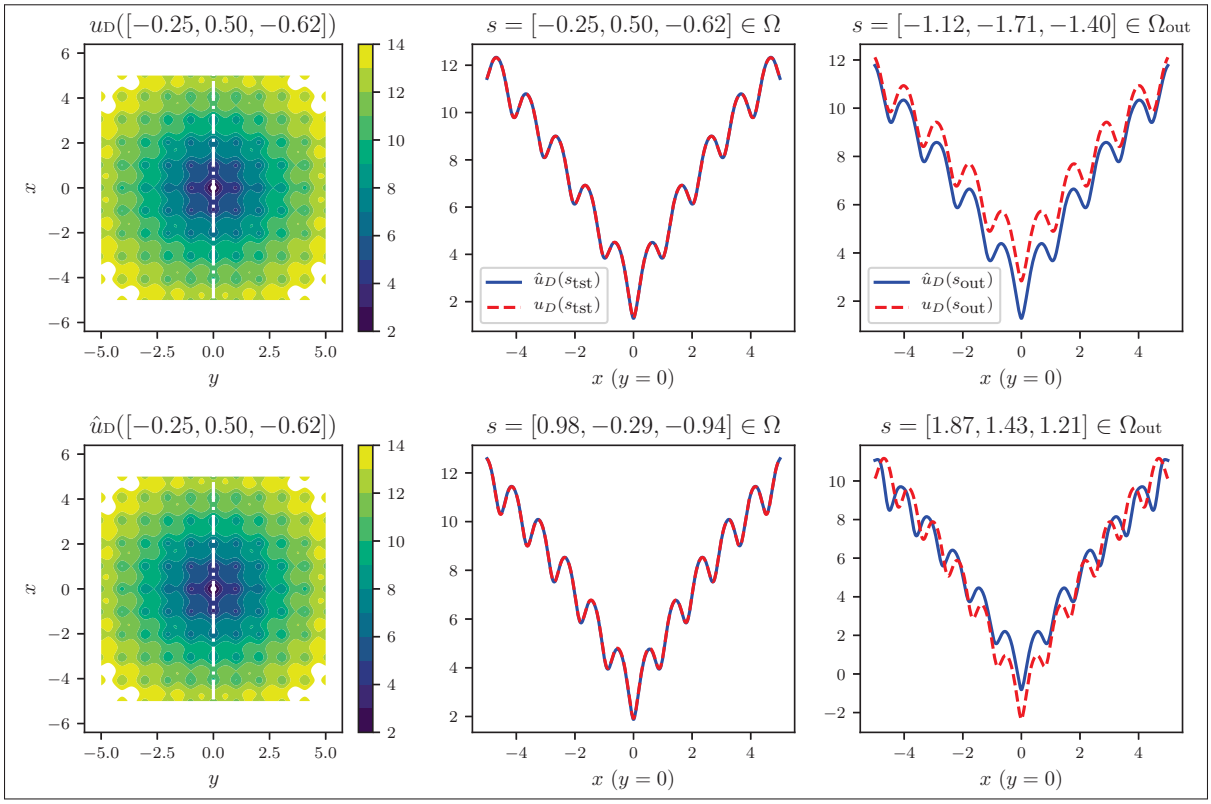


Figure 2.8 Ackley Function (2D). Quick visualization with contour plots of the first column, with the cross-section $y = 0$ pictured. One compares the predicted $\hat{u}_D$ and the observed data u_D across two random test snapshots vertically on the second column, and on the last one, two samples s_{out}, taken outside the dataset bounds

2.4.2.3 Results

The initial dataset preparation was quasi-instantaneous, using the settings mentioned above on a standard desktop computer, while the training took 1 minute and 31 seconds on a regular desktop CPU. The relative errors reached were $RE_{\text{val}} = 0.11\%$ and $RE_{\text{tst}} = 0.11\%$ as well for validation and testing, respectively. In Figure 2.8, we display the results for four different parameter set, two of them from the testing set on the second (and first) column, as as well as out-of-distribution predictions on the third column, sampled in the domain

$$\Omega_{\text{out}} = [-2, -1] \cup [1, 2]. \tag{2.19}$$

As it was the case with Shekel Function in Section 2.4.1, the performance of the model shown in Figure 2.8 is great on the second column, which represents three samples of the test dataset, while it's drastically decreasing on the third column, which represents samples taken out-of-distribution.

2.4.3 Burgers' equation solution (1D, time-dependent)

2.4.3.1 Definition

As a third benchmark, let's introduce the solution of the viscous Burgers' equation—commonly used as a first step before the more complex Navier Stokes equations, yet notoriously hard to work with for computational methods because of its shock-forming behavior, Raissi *et al.* (2019a). It can take one stochastic parameter, the fluid viscosity, denoted here as s. Being real-valued ($D = 1$) and one-dimensional in space, it is defined as

$$\begin{aligned} u : \mathbb{R}^{2+1} &\to \mathbb{R} \\ (x, t; s) &\mapsto \tilde{u}(x, t; s), \end{aligned} \tag{2.20}$$

with the non-spatial parameters vector $\boldsymbol{s} = s$ of size $P = 1$, and $\tilde{u}(x, t; \nu)$ being an analytically available solution of the following PDE—Burgers' equation with an initial sine condition, as presented in Basdevant, Deville, Haldenwang, Lacroix, Ouazzani, Peyret, Orlandi & Patera (1986), with the subscripts denoting the partial derivatives

$$u_t + uu_x - su_{xx} = 0, \quad x \in \Omega_x = [0, 1.5], \quad t \in \Omega_t = [1, 5], \tag{2.21}$$

$$\begin{aligned} u(0, t) = u(1.5, t) = 0, \quad 1 \leq t, \\ u(x, 1) = \frac{x}{1 + \exp\left[\frac{1}{4s}(x^2 - \frac{1}{4})\right]}, \quad 0 < x < 1.5. \end{aligned} \tag{2.22}$$

There exists a directly available analytical solution according to Maleewong & Sirisup (2011), expressed as

$$\tilde{u}(x,t,s) = \frac{x/t}{1 + (t/t_0)^{1/2} \exp\left(\frac{x^2}{4st}\right)}, \quad 1 \leq t, \tag{2.23}$$

with $t_0 = \exp(1/8s)$.

2.4.3.2 Setup

The 1D space domain $\Omega_x = [0, 1.5]$ is discretized uniformly in $N_{x_1} = N_x = 256$, leading the number of DOFs to be $H = 256$.

With $N_t = 100$ time-steps in the time domain $\Omega_t = [1, 5]$, we generate $N_s = 100$ samples of the parameters $\boldsymbol{s}$ using an LHS strategy over the domain $\Omega_s = [0.001, 0.010]$ and produce the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times N_s N_t}$. This domain is chosen to be spread around the value of $s = 0.005$ used in Maleewong & Sirisup (2011) and similar to $s = 0.01/\pi$ in Raissi *et al.* (2019a).

After getting the reduced POD bases via (2.4–2.8) and the dual-step POD approach, picking $\epsilon = 10^{-10}$ and $\epsilon_0 = 10^{-8}$, and generating a full set, we use our default (80%, 20%) *train/validation* ratio to split it into $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{\nu}\}$ of size $N = 80 * 100$, $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{\nu}_{\text{val}}\}$ of size $N_{\text{val}} = 20 * 100$, and $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{\nu}_{\text{tst}}\}$ of size $N_{\text{tst}} = 100 * 100$.

We choose a fixed learning rate of $\eta = 0.005$ for the Adam optimizer, as well as an L2 regularization with $\lambda = 10^{-8}$. No mini-batch split is performed. The training epochs number is set to $N_e = 30,000$.

2.4.3.3 Results

Using the settings mentioned above, on a standard desktop computer, the initial dataset preparation was quasi-instantaneous, while the training took 1 minute and 28 seconds on an NVIDIA Tesla V100 GPU. The dataset is larger because of the time dimension, so the training has indeed been performed on a GPU rather than a desktop CPU, as for the two previous benchmarks. The relative errors reached were $RE_{\text{val}} = 1.37\%$ and $RE_{\text{tst}} = 1.36\%$ for validation and testing, respectively.

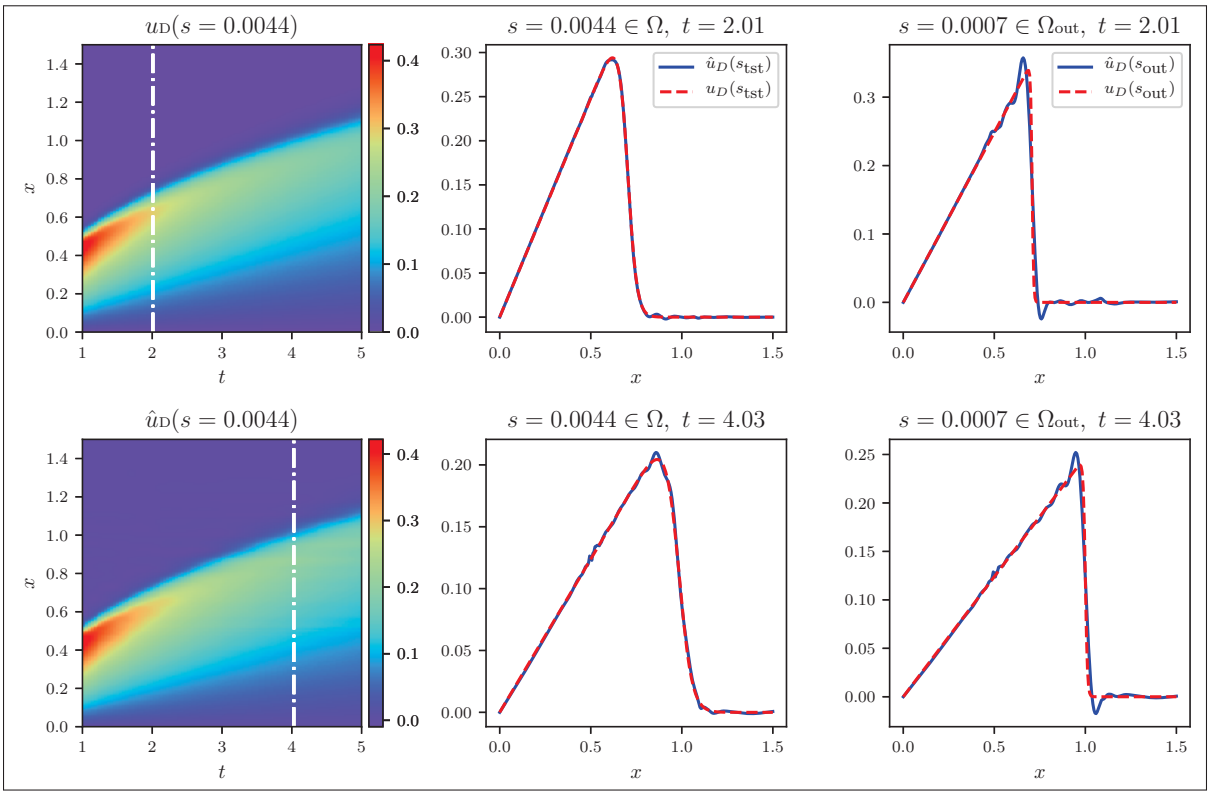


Figure 2.9 Burgers' equation (1D, unsteady). As a quick visualization, one can see colormaps of a random test sample of the first column, as well as the time-steps depicted by the white lines. Then, from left to right: comparing the predicted $\hat{u}_D$ and the analytical data u_D from the dataset at the time-steps, and across two random snapshots for the viscosity parameter s, respectively in and out of the training bounds

For a quick visualization of the solutions, Figure 2.9 pictures the true value and the predicted solution for a random test sample in a colormap on the first column. The second column shows the same sample at the two different time-steps depicted as the white lines on the colormaps. The third column is home to out-of-distribution predictions, for one parameter sampled in the domain

$$\Omega_{\text{out}} = [0.0005, 0.001] \cup [0.010, 0.0105] \tag{2.24}$$

While the predictions made out-of-distribution in the last column of Figure 2.9 are still decreasing in accuracy versus the ones made within the dataset bounds (second column) on the different

time snapshots, we notice instabilities appearing around the shockwave, on both the test samples (less noticeable) and the out-of-distribution samples (more noticeable). We will see in Section 4.4.2 that a Negative Log-Likelihood trained network is handling it better, and the contribution of adversarial training, first introduced by Goodfellow *et al.* (2014a) and used in Lakshminarayanan *et al.* (2017), will also be of help.

2.5 Concluding remarks on the POD-NN framework

Applying the POD-NN framework to these three benchmarks has helped us achieve our Objective 2., creating a fast surrogate model, and Objective 1., a general one since the benchmarks were of dimensions 1D and 2D, and of various time dependence.

The offline-online POD-NN paradigm presented by Hesthaven & Ubbiali (2018), and extended for time-dependent problems in Wang *et al.* (2019) helped us to achieve exactly that, with reasonable accuracy and flexibility: we have indeed used the same architecture across all benchmarks.

We note that a sample benchmark code has only been attached for the Shekel case (Figure 2.7) since it is only a matter of tuning a few hyperparameters and the definition of the function u for the framework to run correctly on the others.

CHAPTER 3

UNCERTAINTY QUANTIFICATION IN DEEP NEURAL NETWORKS

Even though the previous chapter has fulfilled some of our expectations, putting together a fast surrogate model that would be able to make real-time predictions without going back to the mathematical equations (Objective 1., 2.), we still don't know anything about the *confidence* we can have in the predictions (Objective 3.). That is what this chapter is attempting to develop, and is the basis of our contribution to the POD-NN framework.

In Chapter 2, our Neural Network's prediction can indeed be thought of as a probability $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w})$, with $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$ denoting our dataset. Yet, one can't know anything about what this distribution is since we only retrieve its mean, $\hat{u}_D$, which is nothing but a *point estimate*. There have been, however, multiple attempts to track uncertainties within a Deep Neural Network. In this chapter, we will present two of them to build a model that can provide a prediction, and its associated uncertainties, especially some indication on the outputs should be provided if the inputs are out of the domain of learning. In other words, that *knows when it doesn't know*.

3.1 Different types of uncertainty

When one thinks of uncertainties arising from the use of a model, in our case, a Deep Neural Network, there could be multiple ways to think about it. Yet, the most common in the literature is the dichotomy into two groups: *aleatoric* and *epistemic*.

The first one refers to the inherent randomness in the data we observe. No matter how good our model gets, if the information is issued from any measurement, there must be some kind of noise around it, and it can't be reduced, even with additional data. One way to think about it is a fair coin toss. No matter how many times we observe the experiment, we won't be able to make predictions that go below the inherent probability, as explained in Keydana (2019).

The latter is directly linked to the model, and is, in contrary to the aleatoric uncertainty, prone to decreasing with additional data. In a perfect world with an ideal model, this uncertainty would be null.

As we go forward in this chapter, we will try to add an uncertainty component to our Deep Neural Network regression, that addresses both of these categories. It is, however, interesting to

note that in our case, the epistemic uncertainty is very valuable, since our Objective 3. is about getting a warning for out-of-distribution predictions. This uncertainty is tightly linked to how the model fits the data. In a data-free domain, there's an infinity of ways for a model to predict, leading to sizeable epistemic uncertainty.

3.2 Deep Ensembles

3.2.1 Definition

As demonstrated in the paper Lakshminarayanan *et al.* (2017), there is a simple and straightforward way to enable Deep Neural Networks to take care of their associated uncertainties. It is achieved by having them provide a trainable variance output in addition to the regular mean, and train them multiple times with random initialization of their parameters, to average over their different outcomes, as it is depicted in Figure 3.1. The idea of the dual output was first introduced in Nix & Weigend (1994). The concept of ensembles isn't exactly new either and has been detailed extensively in Goodfellow *et al.* (2016), yet using them as a way to quantify uncertainties is the contribution of Lakshminarayanan *et al.* (2017). We see in Figure 3.1 that the number of outputs is indeed doubled from a simple one-dimensional case like in Section 1.4.2.5. It's a consequence of the choice to output μ_y and ρ_y, which are respectively denoting the predicted mean of the output variable, y, and a *raw variance*.

A simple way to think about this captured aleatoric uncertainty would be a measurement problem, where one wants to retrieve some quantity u, yet would retrieve some additional non-constant noise n on top of it, resulting in $u(x) + n(x)$ for any parameter x of the measurable domain. Having the dual-output within the NN of a mean and a variance would directly try to model these components during the regression process.

In this context, the final layer raw variance node(s) will feature a *softplus* activation function, defined as

$$\text{softplus}(x) = \log(1 + \exp(x)), \tag{3.1}$$

which will help to ensure the positivity of the real predicted variance $\sigma_y^2 = \text{softplus}(\rho_y^2)$.

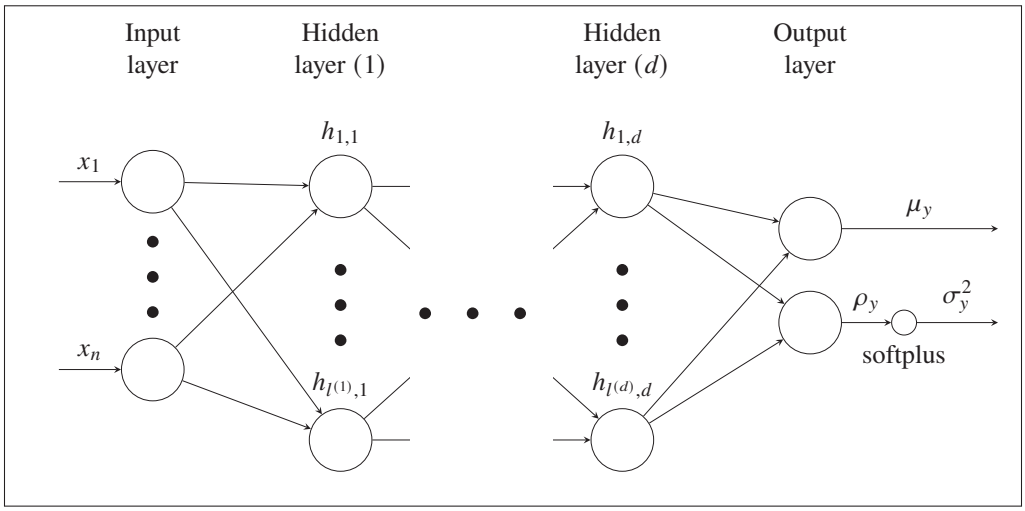


Figure 3.1 $\hat{u}(\boldsymbol{x};\boldsymbol{w}) = \mathcal{N}(\mu_y|0,\sigma_y^2)$, a Deep Neural Network with a dual-output

3.2.2 Training

A custom loss function is being used, a negative log-likelihood (NLL) due for optimization w.r.t. the parameters $\boldsymbol{\theta} = (\boldsymbol{w}, \boldsymbol{b})$ of the network. If one indeed makes the assumption of normally distributed errors around the prediction $\hat{u}(x)$ with a variance $\sigma(x)$, for a data point (x, y) in the dataset, one can write the probability of seeing the data as

$$p(y|x) = \frac{1}{\sqrt{2\pi\sigma^2(x)}} \exp\left(-\frac{(y-\hat{u}(x))^2}{2\sigma^2(x)}\right). \tag{3.2}$$

If one takes the opposite log on each side, it writes as a summation, Nix & Weigend (1994),

$$-\log p(y|x) = \frac{\log 2\pi}{2} + \frac{\log \sigma^2(x)}{2} + \frac{(y-\hat{u}(x))^2}{2\sigma^2(x)}, \tag{3.3}$$

which, applied to the setting of a $\boldsymbol{\theta}$-parametrized network and a minimization context where the constant $\log(2\pi)/2$ won't play a role, considering all the datapoints at once, is constructing the following loss $\mathcal{L}_{NLL}$ defined as

$$\mathcal{L}_{\text{NLL}}(\boldsymbol{\mathcal{D}}, \boldsymbol{\theta}) := \frac{\log\ \boldsymbol{\sigma}_{\boldsymbol{\theta}}^2(\boldsymbol{x})}{2} + \frac{(\boldsymbol{y}-\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}))^2}{2\boldsymbol{\sigma}_{\boldsymbol{\theta}}^2(\boldsymbol{x})}, \tag{3.4}$$

with $(\boldsymbol{\mu_\theta}, \boldsymbol{\sigma_\theta})$ being respectively the mean and the variance, the two outputs of this specifically designed NN, in the case of a dataset $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$. This loss, compared to a standard MSE, allows for direct uncertainties capturing and has been designed by the authors as a *proper scoring rule* which performed well in their experiments Lakshminarayanan *et al.* (2017).

The activation function used by the authors is the ReLU nonlinearity, defined as $\phi(x) = \max(0, x)$, as presented in Section 1.4.2.5.

As noted by the authors, state-of-the-arts results can be achieved training only $M = 5$ randomly initialized networks, which is not a big overhead, especially considering our relatively small training times, as discussed in Section 2.4. This initialization is performed in a *glorot uniform*, or *Xavier* way, Glorot & Bengio (2010), and picks the initial weights of the network in a uniform distribution. For a layer j, with $l^{(j-1)}$ and $l^{(j)}$ denoting the width of the previous layer and the width of the layer j, we, therefore, initialize each weight i of this layer j as follows

$$\boldsymbol{w}_j \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{l^{(j-1)} + l^{(j)}}}, \frac{\sqrt{6}}{\sqrt{l^{(j-1)} + l^{(j)}}}\right] \tag{3.5}$$

3.2.3 Predictions

The authors of Deep Ensembles suggest approximating the mixture of each NN outputs $(\mu_{\boldsymbol{\theta_m}}, \sigma_{\boldsymbol{\theta_m}})$ in a single normal distribution $\mathcal{N}(\mu_*, \sigma_*)$, with the mean defined as

$$\mu_* = \frac{1}{M}\sum_{m=1}^{M} \mu_{\boldsymbol{\theta}_m}, \tag{3.6}$$

with the variance subsequently computed as as

$$\sigma_*^2 = \frac{1}{M}\sum_{m=1}^{M} (\sigma_{\boldsymbol{\theta}_m}^2 + \mu_{\boldsymbol{\theta}_m}^2) - \mu_*^2. \tag{3.7}$$

An example implementation will follow in Section 3.4.

3.3 Bayesian Neural Networks

3.3.1 Presentation and main issue

While Deep Ensembles kept a frequentist point of view, there is a different class of networks adopting the second vision to probabilities, for which the basics have been presented through a curve-fitting example in Section 1.4.2.3. In this section, we will introduce the concept of Variational Inference to workaround the analytical intractabilities of fully Bayesian approaches to create trainable Bayesian Neural Networks, as shown in Blundell *et al.* (2015).

With $\mathcal{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$ as our N-sized dataset, we wish to construct a *likelihood function* $p(\mathcal{D}|\boldsymbol{w})$, tied to our weights and biases of our model, reduced to $\boldsymbol{w}$ for clarity. In a Bayesian context, we think of a *prior* distribution $p(\boldsymbol{w})$—that is, the knowledge we decide to constrain the distribution on the weights with, and Bayes theorem provides us with the *posterior* distribution $p(\boldsymbol{w}|\mathcal{D})$, i.e., the new distribution of $\boldsymbol{w}$ after the dataset is observed, which is defined proportionally with regards to our likelihood and prior as following

$$p(\boldsymbol{w}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w}). \tag{3.8}$$

It is interesting to note that this right-hand side can be maximized and represents the Maximum A Posteriori, which, in a similar fashion to a regular Mean Squared Error, only deals with mean values, and doesn't offer a full distribution over all possible models parametrized by some weights $\boldsymbol{w}$. This case is covered in Section 1.4.2.4. To achieve the fully Bayesian treatment, one needs to deal with the *posterior predictive distribution* for a new pair (x, y) outside of $\mathcal{D}$ instead, expressed as

$$p(y|x, \mathcal{D}) = \int p(y|x, \boldsymbol{w})p(\boldsymbol{w}|\mathcal{D})d\boldsymbol{w}. \tag{3.9}$$

The goal here is indeed to provide an entire distribution for the outputs, rather than a point estimate. Unfortunately, the posterior $p(\boldsymbol{w}|\mathcal{D})$ and, therefore, this integral is intractable in a Neural Network context since it would have to account for every possible weights configuration, which would mean using an infinite number of models, as explained in Blundell *et al.* (2015).

3.3.2 Workaround: Variational Inference

A way for this issue to be addressed is known as Variational Inference and involves an approximation of the true posterior $p(\boldsymbol{w}|\mathcal{D})$ with a new $\boldsymbol{\theta}$-parametrized distribution $q(\boldsymbol{w}|\boldsymbol{\theta})$ of a known form.

The Kullback-Leibler divergence KL is introduced to measure the difference between two distributions $P(x)$ and $Q(x)$, and is defined as

$$\mathrm{KL}(P(x)||Q(x)) = \int P(x) \log \frac{P(x)}{Q(x)}\, d\boldsymbol{x}. \tag{3.10}$$

In our case, the goal is then to minimize this difference measure w.r.t. to the parameters $\boldsymbol{\theta}$, defined as

$$\mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta})||p(\boldsymbol{w}|\mathcal{D})) = \int q(\boldsymbol{w}|\boldsymbol{\theta}) \log \frac{q(\boldsymbol{w}|\boldsymbol{\theta})}{p(\boldsymbol{w}|\mathcal{D})}\, d\boldsymbol{w} \tag{3.11}$$

Remembering the Bayes rule and $p(\boldsymbol{w}|\mathcal{D}) = p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})/p(\mathcal{D})$, the optimization goal can write as

$$\begin{aligned}
\mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta})||p(\boldsymbol{w}|\mathcal{D})) &= \int q(\boldsymbol{w}|\boldsymbol{\theta}) \log \frac{q(\boldsymbol{w}|\boldsymbol{\theta})p(\mathcal{D})}{p(\boldsymbol{w})p(\mathcal{D}|\boldsymbol{w})}\, d\boldsymbol{w} && (3.12)\\
&= \mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta})||p(\boldsymbol{w})) - \mathbb{E}_{q(\boldsymbol{w}|\boldsymbol{\theta})} \log p(\mathcal{D}|\boldsymbol{w}) + \log p(\mathcal{D}) && (3.13)\\
&=: \mathcal{F}(\mathcal{D}, \boldsymbol{\theta}) + \log p(\mathcal{D}). && (3.14)
\end{aligned}$$

This first term, $\mathcal{F}(\mathcal{D}, \boldsymbol{\theta})$, is usually known in the literature as the *variational free energy* and is equivalent to minimizing the KL w.r.t. $\boldsymbol{\theta}$ since the other term $\log p(\mathcal{D})$ doesn't depend on $\boldsymbol{\theta}$. The variational free energy is a sum of two terms, the first being linked to the prior, named *complexity cost*, while the latter is related to the data and referred to in Blundell *et al.* (2015) as the *likelihood cost*. It is shown to be approximated by drawing N_{mc} samples $\boldsymbol{w}_m$ from $q(\boldsymbol{w}|\boldsymbol{\theta})$ at the layer level, and N samples at the output level (for each training input), as follows

$$\mathcal{F}(\mathcal{D}, \boldsymbol{\theta}) \approx \sum_{m=1}^{N_{\mathrm{mc}}} [\log q(\boldsymbol{w}_m|\boldsymbol{\theta}) - \log p(\boldsymbol{w}_m)] - \sum_{m=1}^{N} \log p(\mathcal{D}|\boldsymbol{w}_m). \tag{3.15}$$

Another term one can often see in the literature is the Evidence Lower Bound (ELBO), that is defined as the opposite of $\mathcal{F}(\mathcal{D}|\boldsymbol{\theta})$, and due to the positivity of the KL, it is effectively a lower bound on $\log p(\mathcal{D})$. We, therefore, denote the corresponding minimization objective, our loss function in practice, by $\mathcal{L}_{\text{ELBO}}(\mathcal{D}, \boldsymbol{\theta}) := \tilde{\mathcal{F}}(\mathcal{D}, \boldsymbol{\theta})$, with $\tilde{\mathcal{F}}$ representing the approximation made in (3.15).

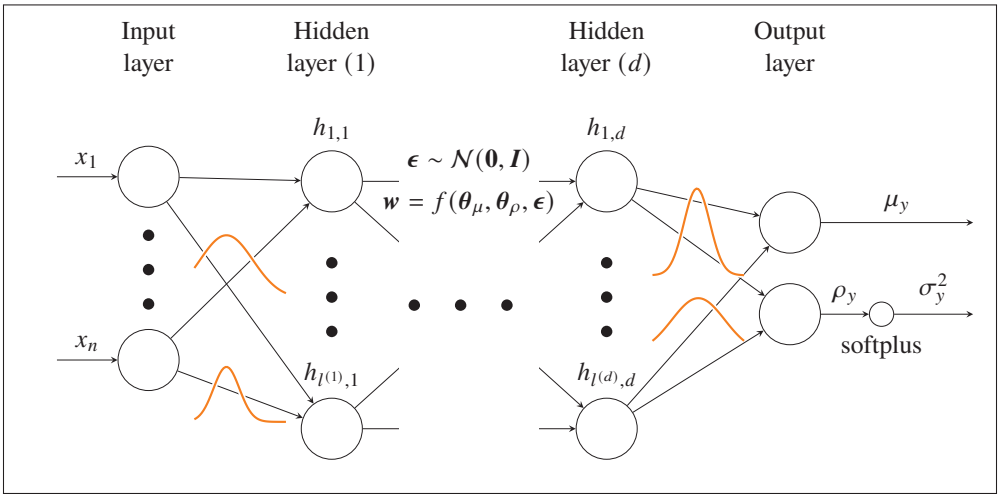


Figure 3.2 $\hat{u}(\boldsymbol{x}; \boldsymbol{w}) = \mathcal{N}(\mu_y|0, \sigma_y^2)$, a probabilistic Bayesian Neural Network, with distributions on the weights, and a dual-ouput

3.3.3 Training and predictions

In a similar fashion to Section 2.3, the model is trained using the backpropagation of the gradients. Figure 3.2 shows a representation of a Bayesian Neural Network, with distributions on the weights, as well as a dual mean and variance output. The variational posterior for each weight $\boldsymbol{w}^{(j)}$, or bias $\boldsymbol{b}^{(j)}$, is parametrized by the trainable parameters $\boldsymbol{\theta}^{(j)} = (\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\rho^{(j)})$. They correspond to a mean and a raw variance, which are randomly initialized at first. This setting grows the number of total parameters to be twice the one in a regular NN.

3.3.3.1 Reparametrization trick

In this context, the reparametrization trick presented in Kingma & Welling (2014) is needed for the posterior: at the j-th layer level, a random noise from *parameter-free* distribution is sampled,

for instance, $\boldsymbol{\epsilon}^{(j)} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, with $\mathbf{0}$ and $\boldsymbol{I}$ denoting the null and identity matrices of the right size.

We then construct a determinist and differentiable function, such as $f(\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\rho^{(j)}, \boldsymbol{\epsilon}^{(j)}) = \boldsymbol{\theta}_\mu^{(j)} + \boldsymbol{\theta}_\rho^{(j)} \odot \boldsymbol{\epsilon}^{(j)}$, with $\odot$ denoting the element-wise multiplication, which will be used at the backpropagation step.

3.3.3.2 Training workflow

As presented in Blundell *et al.* (2015), the workflow is the following, for each training epoch.

1. For each layer j:
   a. Sample a random noise $\boldsymbol{\epsilon}^{(j)}$ for the weights and the biases,
   b. Initialize the weights and the biases, $\boldsymbol{w}^{(j)} = f(\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\rho^{(j)}, \boldsymbol{\epsilon}^{(j)})$,
   c. Compute each processed variance $\boldsymbol{\theta}_\sigma^{(j)} = \text{softplus}(\boldsymbol{\theta}_\rho^{(j)})$,
   d. Add the layer contribution to the loss function $\mathcal{L}_{\text{ELBO}}$, with $q(\boldsymbol{w}^{(j)}|\boldsymbol{\theta}^{(j)}) = \mathcal{N}(\boldsymbol{w}^{(j)}|\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\sigma^{(j)})$, minus the log of the prior, $\log p(\boldsymbol{w}^{(j)})$.
2. The last term of the loss is added from the outputs of the network (μ_y, σ_y^2) and is, in fact, a Negative Log-Likelihood contributed to the loss, computed from a Normal distribution of the output as
$$-\log p(\mathcal{D}|\boldsymbol{w}) = \mathcal{N}(\boldsymbol{v}|\mu_y, \sigma_y^2). \tag{3.16}$$
3. The regular backpropagation step can finally take place, finishing the epoch by updating each parameter with the derivative of the fully reconstructed loss $\mathcal{L}_{\text{ELBO}}$ w.r.t. the parameters $\boldsymbol{\theta}$.

One can note that the Monte-Carlo approximation made in (3.15) for the loss function is taking place with N_{mc} being the number of parameters in the network since a configuration $\boldsymbol{w}_m$ is triggered while going through each of them, and the sum of the first two terms is done iteratively while going through the network. However, the NLL sum term is added from the outputs (Step 2. above), and in that case, N represents the number of points in the dataset.

Each hidden state j is computed as usual in a feedforward NN,

$$\boldsymbol{h}^{(j)} = \phi\left(\boldsymbol{w}^{(j)}\boldsymbol{h}^{(j-1)} + \boldsymbol{b}^{(j)}\right), \tag{3.17}$$

with a nonlinearity ϕ, yet each weight and bias computed via the parameters we actually train, $(\boldsymbol{\theta}_{\mu}^{(j)}, \boldsymbol{\theta}_{\sigma}^{(j)})$, and a sample noise $\boldsymbol{\epsilon}^{(j)}$, as seen above in Step 1.

3.3.3.3 Predictions

This is what makes the whole network probabilistic since each forward pass has a component of randomness within, contributing to the epistemic uncertainty.

Its non-deterministic predictions have to be evaluated B times to effectively get an approximation of the intractable predictive posterior in (3.9), and, therefore, grasp the epistemic side of the related uncertainties. These B predictions will then be averaged over like in (3.6) and (3.7) to reach the mean μ_* and variance σ_*^2, as

$$\mu_* = \frac{1}{B}\sum_{b=1}^{B} \mu_{\boldsymbol{\theta}_b}, \tag{3.18}$$

$$\sigma_*^2 = \frac{1}{B}\sum_{b=1}^{B} (\sigma_{\boldsymbol{\theta}_b}^2 + \mu_{\boldsymbol{\theta}_b}^2) - \mu_*^2. \tag{3.19}$$

After the training phase of this unique network, one chooses a number of samples B for the prediction estimation, that can be large since the evaluation phase of a network is very fast. In practice $B = 100$ has been often used, but as for the Deep Ensembles, going as low as $B = M = 5$ should encompass enough information.

3.3.3.4 Prior and initialization details

For the prior, one can choose to either have it *trainable*, for instance by having the parameters of a Normal distribution within the parameters of the network $\boldsymbol{\theta}$, or predefined, and the choice made by the authors of Blundell *et al.* (2015) is a scale mixture defined as

$$p(\boldsymbol{w}) = \pi_0 \mathcal{N}(\boldsymbol{w}|0, \pi_1^2) + (1-\pi)\mathcal{N}(\boldsymbol{w}|0, \pi_2^2), \tag{3.20}$$

with π, π_1 and π_2 hyperparameters to be set by the user, with the conditions $\pi_1 > \pi_2$ and $\pi_2 \ll 1$, according to Blundell *et al.* (2015). In practice, we've used continuously $\pi_2 = 0.1$, and $\pi_0 = \frac{1}{2}$, as in Krasser (2019), yet multiple values for π_1, such as 1.5, 3, and 4, since it appeared tightly linked to the predicted uncertainties, especially out-of-distribution.

Each trainable parameter $\boldsymbol{\theta}^{(j)}$ (weight or bias) of the j-th layer can be randomly initialized, with

$$\boldsymbol{\theta}^{(j)} = (\boldsymbol{\theta}_{\mu}^{(j)}, \boldsymbol{\theta}_{\sigma}^{(j)}) \sim \mathcal{N}\left(\mathbf{0}, \sqrt{\pi_0 \pi_1^2 + (1-\pi_0)\pi_2^2}\boldsymbol{I}\right). \tag{3.21}$$

An example implementation with sample Python code will be presented in Section 3.4 and specifically in Figure 3.9. Additionally, a pseudo-code implementation for the flood modeling application is provided later in Algorithm 4.3.

Table 3.1 Handling of uncertainties for the two chosen approaches: Bayesian Neural Networks and Deep Ensembles

	BNN	*EnsNN*
Aleatoric σ_{al}	Multi-output (μ, ρ) $\sigma_{al} = \log(1 + e^{\rho})$	Multi-output (μ, ρ) $\sigma_{al} = \log(1 + e^{\rho})$
Epistemic σ_{ep}	Built-in with the distribution on $(\boldsymbol{w}, \boldsymbol{b})$ Train: 1x, Predict: B times	Random init of $(\boldsymbol{w}, \boldsymbol{b})$ Train + Predict: M times

3.4 Summary and tests

Let's consider an effortless setup to assess the quality of the methods above, the same as in Section 1.4.2.2, that was borrowed from Lakshminarayanan *et al.* (2017). The focus is on the cubic function $u(x) = x^3$, acting as our exact "unknown" solution. We sample $N = 20$ points within the training domain $x \in [-4, 4]$, with an artificial Gaussian noise of $\sigma_{al} = 9$, representing the aleatoric uncertainty. After training the model $\hat{u}$, we will use it to make predictions over a larger domain $x \in [-6, 6]$, with $N_{\text{tst}} = 300$, to see how our model performs when the inputs are *out-of-distribution*.

For all tests, the library TensorFlow, Abadi (2016), has been used in version 2.1.0 to define and train regular Deep Neural Networks using *Dense* layers, and the module TensorFlow Probability which provides distributions. It allows for a natural definition of custom *DenseVariational* layers, adapted from a blog post, Krasser (2019), dedicated to the implementation of Blundell *et al.* (2015).

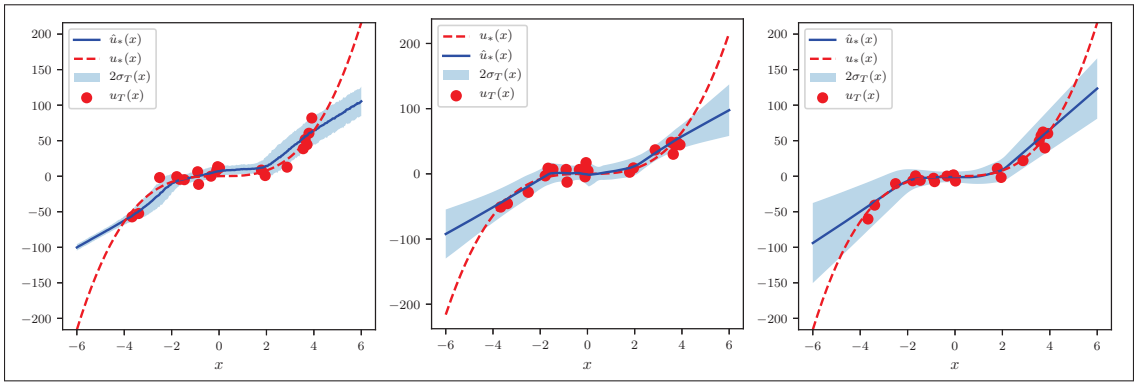


Figure 3.3 Toy problem. Curve-fitting of a cubic function. From left to right, UQPINNs (without the physics-informed loss), Deep Ensembles, and Bayesian Neural Networks are used

The results are pictured in Figure 3.3. While the uncertainty quantification is present in the UQPINNs outcome, the fact that the confidence interval around the predicted mean goes to zero as we exit the training domain prevents us from using it. This model, quickly presented in Section 1.4.3.3, wouldn't indeed allow us to realize our Objective 3., having a model that knows when it doesn't. A tentative explanation for this would be that the authors of the UQPINNs paper worked on PDEs with boundary conditions. Hence, they were probably concerned about what was happening within those boundaries and not on the outside.

We can, however, observe that both the Deep Ensembles and the Bayesian Neural Network models perform as expected, with uncertainties increasing on both ends.

Additionally, Table 3.1 is introduced to quickly sum up the primary difference in the handling of epistemic uncertainty in Deep Ensembles and Bayesian Neural Networks.

Code snippets for the Deep Ensembles model and the Bayesian Neural Network model applied to this toy problem can be found, respectively, in Figure 3.7 and 3.5. They make use of two classes, *VarNeuralNetwork* and *BayesianNeuralNetwork*, that are also respectively depicted by sample Python code in Figure 3.10 and 3.8. The custom Keras layer *DenseVariational* that is being used in the latter can be found in Figure 3.9.

It is also to be noted that, in the Deep Ensembles case, the M steps of the training can be easily distributed on different devices—provided it isn't an issue, and therefore reduce the overhead.

To achieve this, we've used the Horovod library, Sergeev & Del Balso (2018), which provides an easy interface to control the devices that TensorFlow sees. A sample code of its implementation on the toy problem is shown in Figure 3.6.

To illustrate the two types of uncertainties, we've drawn three samples from the trained BNN, and have plotted them in Figure 3.4. One can see three dark blue lines, representing the mean of each prediction, and two associated light blue lines, located at ±2 standard deviations given by each prediction. These errors represent the aleatoric uncertainty, while combining these predictions in a mixture as in (3.7) gives rise to the epistemic uncertainty, which grows bigger when one exits the training bounds.

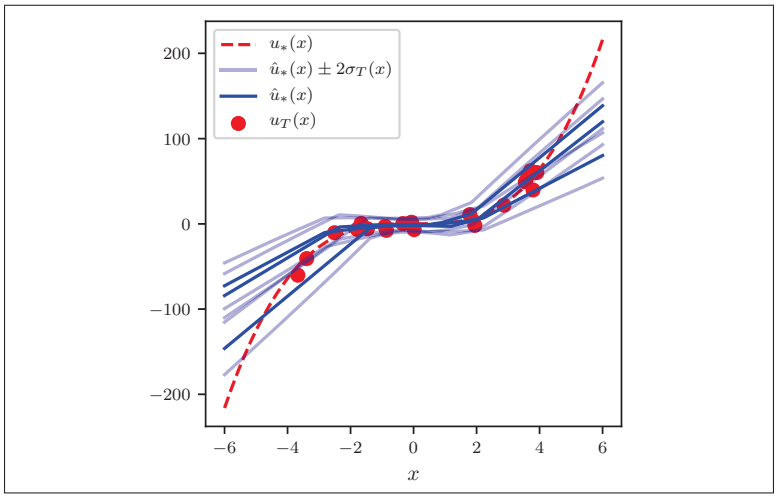


Figure 3.4 Toy problem. 3 samples drawn from a BNN model, with their respective aleatoric uncertainty

```
from podnn.custombnn import BayesianNeuralNetwork

layers = [1, 20, 20, 1]
model = BayesianNeuralNetwork(layers, lr=0.05, klw=1., soft_0=1.,
                              sigma_alea=noise_std,
                              adv_eps=None, norm="minmax")
model.fit(x, y, epochs=15000, batch_size=batch_size)
u_pred, u_pred_var = model.predict(x_tst)
u_pred_sig = np.sqrt(u_pred_var)
```

Figure 3.5 Sample of code for the BNN toy problem

```
import numpy as np
from podnn.varneuralnetwork import VarNeuralNetwork

# Setting up the Horovod library
import horovod.tensorflow as hvd
tf.config.set_soft_device_placement(True)
hvd.init()
gpu_id = hvd.local_rank()
phys_devices = tf.config.experimental.get_visible_devices('GPU')
tf.config.experimental.set_visible_devices(phys_devices[gpu_id], 'GPU')

# Will be run on the hvd.local_rank() device
layers = [1, 20, 20, 1]
model = VarNeuralNetwork(layers, lr=0.01, lam=0.001, norm="minmax")
model.fit(x, y, epochs=5000)
model.save()

# To be run with this command for M=5 GPUs (on one node)
# horovodrun -np 5 -H localhost:5 python train.py
```

Figure 3.6 Distributed version of the Deep Ensembles toy problem code

```
import numpy as np
from podnn.varneuralnetwork import VarNeuralNetwork

layers = [1, 20, 20, 1]
M = 5
u_pred_samples = np.zeros((M, y_tst.shape[0], y_tst.shape[1]))
u_pred_var_samples = np.zeros_like(u_pred_samples)

for i in range(5):
    model = VarNeuralNetwork(layers, lr=0.01, lam=0.001, norm="minmax")
    model.fit(x, y, epochs=5000)
    u_pred_samples[i], u_pred_var_samples[i] = model.predict(x_tst)

u_pred = u_pred_samples.mean(0)
u_pred_var = (u_pred_var_samples + u_pred_samples ** 2).mean(0) \
             - u_pred ** 2
u_pred_sig = np.sqrt(u_pred_var)
```

Figure 3.7 Sample of code for the Deep Ensembles toy problem

```
import tensorflow as tf
import tensorflow_probability as tfp
from .podnn.densevar import DenseVariational

class BayesianNeuralNetwork:
    def __init__(self, layers, lr, klw, soft_0, sigma_alea, adv_eps,
                 norm=NORM_NONE, model=None, norm_bounds=None):
        # ...
        self.model = self.build_model()

    def build_model(self):
        n_L = self.layers[-1]
        model = tfk.models.Sequential([
            tfk.layers.InputLayer(self.layers[0]),
            *[
            DenseVariational(
                units=width, activation="relu",
                kl_weight=self.klw, dtype=self.dtype,
            ) for width in self.layers[1:-1]],
            DenseVariational(
                units=n_L, activation="linear",
                dtype=self.dtype, kl_weight=self.klw,
            ),
        ])
        def neg_log_likelihood(y_obs, y_pred, sigma=self.sigma_alea):
            dist = tfp.distributions.Normal(loc=y_pred, scale=sigma)
            return K.sum(-dist.log_prob(y_obs))
        model.compile(loss=neg_log_likelihood,
                      optimizer=tfk.optimizers.Adam(self.lr))
        return model

    def fit(self, X_v, v, epochs, logger, batch_size):
        # ...
        self.set_normalize_bounds(X_v)
        X_v = self.normalize(X_v)
        v = self.tensor(v)
        self.model.fit(X_v, v, epochs=epochs, batch_size=batch_size)

    def predict(self, X):
        X = self.normalize(X)
        return self.model(X)

    def save_to(self, model_path, params_path):
        # ...
        tf.keras.models.save_model(self.model, model_path)

    @classmethod
    def load_from(cls, model_path, params_path):
        # ...
        model = tf.keras.models.load_model(model_path)
        return cls(model=model)
```

Figure 3.8 Sample of code for the *BayesianNeuralNetwork* class

```
import tensorflow as tf
import tensorflow_probability as tfp
tfk = tf.keras
K = tf.keras.backend
tfp = tfp.distributions

class DenseVariational(tfk.layers.Layer):
    def __init__(self, units, kl_weight, activation=None,
                 prior_sigma_1=1.5, prior_sigma_2=0.1,
                 prior_pi=0.5, **kwargs):
        self.units = units #...
        self.sig_i = np.sqrt(prior_pi_1 * prior_sigma_1 ** 2 +
                             prior_pi_2 * prior_sigma_2 ** 2)
        super().__init__(**kwargs)

    def build(self, input_shape):
        self.kernel_mu = self.add_weight(name='kernel_mu',
            shape=(input_shape[1], self.units), trainable=True,
            initializer=tfk.initializers.RandomNormal(stddev=self.sig_i)),
        self.bias_mu = self.add_weight(name='bias_mu',
            shape=(self.units,), trainable=True,
            initializer=tfk.initializers.RandomNormal(stddev=self.sig_i)),
        self.kernel_rho = self.add_weight(name='kernel_rho',
            shape=(input_shape[1], self.units), trainable=True,
            initializer=tfk.initializers.Constant(0.)),
        self.bias_rho = self.add_weight(name='bias_rho',
            shape=(self.units,), trainable=True,
            initializer=tfk.initializers.Constant(0.)),
        super().build(input_shape)

    def call(self, inputs, **kwargs):
        w_sig = 1e-3 + tf.math.softplus(0.1 * self.kernel_rho)
        kernel = self.kernel_mu + w_sig * \
                 tf.random.normal(self.kernel_mu.shape)
        b_sig = 1e-3 + tf.math.softplus(0.1 * self.bias_rho)
        bias = self.bias_mu + b_sig * tf.random.normal(self.bias_mu.shape)

        self.add_loss(self.kl_loss(kernel, self.kernel_mu, kernel_sigma) +
                      self.kl_loss(bias, self.bias_mu, bias_sigma))

        return self.activation(K.dot(inputs, kernel) + bias)

    def kl_loss(self, w, mu, sigma):
        variational_dist = tfp.distributions.Normal(mu, sigma)
        return self.kl_weight * K.sum(variational_dist.log_prob(w) \
               - self.log_prior_prob(w))

    def log_prior_prob(self, w):
        comp_1_dist = tfp.distributions.Normal(0., self.prior_sigma_1)
        comp_2_dist = tfp.distributions.Normal(0., self.prior_sigma_2)
        c = np.log(np.expm1(1.))
        return K.log(c + self.prior_pi_1 * comp_1_dist.prob(w) +
                         self.prior_pi_2 * comp_2_dist.prob(w))
```

Figure 3.9 Sample of code for the custom *DenseVariational* Keras layer

```
import tensorflow as tf

class VarNeuralNetwork:
    # ...
    def build_model(self):
        inputs = tf.keras.Input(shape=(self.layers[0],), name="x")
        for width in self.layers[1:-1]:
            x = tf.keras.layers.Dense(width, activation=tf.nn.relu,
                    kernel_initializer="glorot_normal")(inputs)
        x = tf.keras.layers.Dense(2 * self.layers[-1], activation=None,
                    kernel_initializer="glorot_normal")(x)
        def split_mean_var(data):
            mean, out_var = tf.split(data, num_or_size_splits=2, axis=1)
            var = tf.math.softplus(out_var) + 1e-6
            return [mean, var]
        outputs = tf.keras.layers.Lambda(split_mean_var)(x)
        model = tf.keras.Model(inputs=inputs, outputs=outputs)
        return model

    @tf.function
    def loss(self, y, y_pred):
        y_pred_mean, y_pred_var = y_pred
        return tf.reduce_mean(tf.math.log(y_pred_var) / 2) + \
            tf.reduce_mean((y -  y_pred_mean)**2 / 2*y_pred_var) + \
            self.regularization()

    @tf.function
    def grad(self, X, v):
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(X)
            loss_value = self.loss(v, self.model(X))
            if self.adv_eps is not None:
                loss_x = tape.gradient(loss_value, X)
                X_adv = X + self.adv_eps * tf.math.sign(loss_x)
                loss_value += self.loss(v, self.model(X_adv))
        grads = tape.gradient(loss_value, self.wrap_training_variables())
        del tape
        return loss_value, grads

    def fit(self, X_v, v, epochs, logger):
        # ...
        self.set_normalize_bounds(X_v)
        X_v = self.normalize(X_v)
        v = self.tensor(v)
        for epoch in range(epochs):
            loss_value, grads = self.grad(X_v, v)
            self.tf_optimizer.apply_gradients(
                zip(grads, self.wrap_training_variables()))

    def predict(self, X):
        X = self.normalize(X)
        y_pred_mean, y_pred_var = self.model(X)
        return y_pred_mean.numpy(), y_pred_var.numpy()
```

Figure 3.10 Sample of code for the *VarNeuralNetwork* class

CHAPTER 4

NON-INTRUSIVE REDUCED-ORDER MODELING USING UNCERTAINTY-AWARE DEEP NEURAL NETWORKS AND PROPER ORTHOGONAL DECOMPOSITION: APPLICATION TO FLOOD MODELING

This chapter content is also available as a journal article, which is in the process of being submitted to the Journal of Computational Physics.

4.1 Introduction

Machine Learning and other forms of Artificial Intelligence have been at the epicenter of massive breakthroughs in the notoriously hard fields of computer vision, language modeling, or content generation, such as presented in the work of Szegedy *et al.* (2017), Mikolov *et al.* (2013), and Mikolov *et al.* (2013). Yet, many other fields where robust and heavily-tested methods could be positively impacted by the modern computational tools associated with it: antibiotic discovery is a very recent example of this Stokes, Yang, Swanson, Jin, Cubillos-Ruiz, Donghia, MacNair, French, Carfrae, Bloom-Ackerman, Tran, Chiappino-Pepe, Badran, Andrews, Chory, Church, Brown, Jaakkola, Barzilay & Collins (2020). In the realm of high-fidelity computational mechanics, simulation time is tightly related to the size of the mesh and the number of time-steps, in other words, its accuracy, which might make it impractical to be used in real-time context for new parameters.

Much research has been performed to address this large-size problem and create Reduced-Ordered Models (ROM), that can effectively replace its heavier counterpart for tasks like design and optimization, or real-time predictions. The most common way to build a ROM is to go through a compression phase into a *reduced space*, defined by a set of *reduced basis* (RB), which is at the root of many methods, according to Benner *et al.* (2015). For the most part, RB methods involve an *offline-online* paradigm, where the first is the more computational-heavy one, while the latter should be fast enough to allow for real-time predictions. The idea is to collect data points from simulation, or any high-fidelity source called *snapshots*, and extract the information that has the most significance on the dynamics of the system, the *modes*, via a reduction method in the *offline* stage.

Proper Orthogonal Decomposition, as introduced in Holmes *et al.* (1997); Sirovich (1987), and the algorithm named Singular Value Decomposition (SVD) algorithm, Burkardt *et al.* (2006), is by far the most popular method to reach a *low-rank* approximation. Subsequently, the *online* stage involves recovering the *expansion coefficients*, projecting back into our uncompressed, real-life space. This is where the separation between intrusive and non-intrusive methods appears, where the first is using techniques depending on the problem's formulation, such as the Galerkin procedure, Couplet *et al.* (2005); Zokagoa & Soulaimani (2018). At the same time, the latter tries to statistically infer the mapping by considering the snapshots as a dataset. In this non-intrusive context, the POD-NN framework has been proposed by Hesthaven & Ubbiali (2018) and extended for time-dependent problems in Wang *et al.* (2019), and aims at training an artificial neural network at performing the mapping.

Conventionally, laws of physics are expressed as well-defined PDEs, with boundary/initial conditions as constraints, but lately, pure data-driven methods lead to new approaches in PDE discovery, Brunton *et al.* (2016). The take-off of this new field of Deep Learning in Computational Fluid Dynamics was predicted in Kutz (2017). Its flexibility allows for multiple applications, such as the recovery of missing CFD data in Carlberg *et al.* (2019), or aerodynamic design optimization, Tao & Sun (2019). The cost associated with a fine mesh is high and yet has been overcome with a Machine Learning approach aiming at assessing errors and correcting quantities in a more coarse setting, Hanna *et al.* (2020). New research in the field of numerical schemes has been performed in Després & Jourdren (2020), presenting the Volume of Fluid-Machine Learning (VOF-ML) approach, applied in bi-material settings. A review of the vast landscape of possibilities is explored in Brunton & Kutz (2019). The constraints of *small data* also led researchers to try to balance the need for data in AI contexts with expert knowledge such as the defined governing equations. It was first presented in Raissi *et al.* (2017a), then extended to Neural Networks/hi in Raissi *et al.* (2019a) with applications on Computational Fluid Dynamics, as well as in vibrations Raissi *et al.* (2019b).

While their regression power is impressive, Deep Neural Networks are still, in their standard state, only able to predict a mean value, and don't provide any guidance on how much trust one can put in it. To address this, recent additions to the Machine Learning landscape include

Deep Ensembles, Lakshminarayanan *et al.* (2017), which suggest the training of an ensemble of specific, variance-informed deep neural networks, to get a complete uncertainty treatment. It has been subsequently extended to Sub-Ensembles for faster implementation, Valdenegro-Toro (2019), and later reviewed in Snoek, Ovadia, Fertig, Lakshminarayanan, Nowozin, Sculley, Dillon, Ren & Nado (2019). Prior to this, other works have successfully encompassed the Bayesian view of probabilities within the Deep Neural Network, with the work of Mackay (1995), Barber & Bishop (1998), Graves (2011), Hernandez-Lobato & Adams (2015) ultimately leading to the backpropagation-compatible Bayesian Neural Networks defined in Blundell *et al.* (2015), making use of Variational Inference, Hinton & van Camp (1993), and paving the way for trainable Bayesian Neural Networks, also reviewed in Snoek *et al.* (2019).

In this work, we, therefore, aim at extending the concept of POD-NN with uncertainty quantification in Deep Neural Networks. After going through the methodology of Deep Ensembles, we will test it on two different benchmarks and apply it to flood modeling with the aim of producing probabilistic flooding maps, as well as a dam break scenario, first in a 1D Rieman analytically tractable example, and subsequently in the river setting. As a real application example, we consider the case of the Milles-Iles river in the Montreal, Canada, metro area. Finally, Bayesian Neural Networks are embedded in the framework and applied to one benchmark and the probabilistic flooding maps problem.

4.2 Reduced basis with Proper Orthogonal Decomposition

4.2.1 Objective and setup

We first start by defining u, our $\mathbb{R}^D$-valued function of interest

$$\begin{aligned} u : \mathbb{R}^{n+P} &\to \mathbb{R}^D \\ (\boldsymbol{x}, \boldsymbol{s}) &\mapsto u(\boldsymbol{x}, \boldsymbol{s}), \end{aligned} \tag{4.1}$$

with $\boldsymbol{x} \in \mathbb{R}^n$ the spatial parameters, and $\boldsymbol{s} \in \mathbb{R}^P$ additional non-spatial parameters, anything from a fluid viscosity to the time variable.

Computing this function is costly, so one can only get a finite number of *snapshots* S. These solutions are obtained over a discretized space, which can either be a uniform grid or an unstructured mesh, with n representing its the number of dimensions, and D the total number of nodes. We denote N_s the number of non-spatial parameters sampled, and N_t the number of time-steps considered, which would be greater than one in a time-dependent setting, leading the total of snapshots to be $S = N_s N_t$.

In our applications, the spatial mesh of N_D nodes is considered fixed in time, and since it's known and defined upfront, one can incorporate it in (4.1), removing $\boldsymbol{x}$ as a parameter in u, and making $H = N_D \times D$ the total number of degrees of freedom (DOFs) on the mesh

$$\begin{aligned} u_D : \mathbb{R}^P &\to \mathbb{R}^H \\ \boldsymbol{s} &\mapsto u_D(\boldsymbol{s}). \end{aligned} \tag{4.2}$$

The simulation data, obtained from computing the function u with S parameters, is stored in a matrix of snapshots $\boldsymbol{U} = u_D(\boldsymbol{s}) \in \mathbb{R}^{H \times S}$. Proper Orthogonal Decomposition is used to build a Reduced-Order Model (ROM) and produce a *low-rank approximation*, which will be much more efficient to compute and use when fast multi query simulations are required. With the snapshots method, Sirovich (1987), one can efficiently extract a reduced POD basis in a finite-dimension context. In our case, one starts with the $\boldsymbol{U}$ matrix, and the Singular Value Decomposition algorithm is used, Burkardt *et al.* (2006), to extract $\boldsymbol{W} \in \mathbb{R}^{H \times H}$, $\boldsymbol{Z} \in \mathbb{R}^{S \times S}$, and the r descending-ordered positive singular values matrix $\boldsymbol{D} = \mathrm{diag}(\xi_1, \xi_2, \ldots, \xi_r)$ such as

$$\boldsymbol{U} = \boldsymbol{W} \begin{bmatrix} \boldsymbol{D} & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{Z}^\intercal. \tag{4.3}$$

For the finite truncation of the first L modes, the following criterion on the singular values, with a hyperparameter ϵ

$$\frac{\sum_{l=L+1}^{r} \xi_l^2}{\sum_{l=1}^{r} \xi_l^2} \leq \epsilon, \tag{4.4}$$

and one constructs each reduced vector $\boldsymbol{V}_j \in \mathbb{R}^S$ from $\boldsymbol{U}$ and the j-th column of $\boldsymbol{Z}$, $\boldsymbol{Z}_j$, such as

$$\boldsymbol{V}_j = \frac{1}{\xi_j}\boldsymbol{U}\boldsymbol{Z}_j, \tag{4.5}$$

and finally build our POD basis

$$\boldsymbol{V} = \left[\boldsymbol{V}_1|\ldots|\boldsymbol{V}_j|\ldots|\boldsymbol{V}_L\right] \in \mathbb{R}^{H\times L}. \tag{4.6}$$

4.2.2 Projections

To project from and to the low-rank approximation, one needs projection coefficients, and the ones *corresponding* to the matrix of snapshots are obtained as

$$\boldsymbol{v} = \boldsymbol{V}^\intercal\boldsymbol{U}, \tag{4.7}$$

with $\boldsymbol{U}_{\text{POD}}$ the approximation of $\boldsymbol{U}$, that can be projected back to the expanded space as

$$\boldsymbol{U}_{POD} = \boldsymbol{V}\boldsymbol{V}^\intercal\boldsymbol{U} = \boldsymbol{V}\boldsymbol{v}. \tag{4.8}$$

To assess the quality of the compression/expansion procedure, the relative projection error writes as

$$RE_{POD} = \sum_{j=1}^{S} \frac{||(\boldsymbol{U})_j - (\boldsymbol{U}_{POD})_j||_2}{||(\boldsymbol{U})_j||_2}, \tag{4.9}$$

with the $(\cdot)_j$ subscript denoting the j-th column of the targeted matrix, $||\cdot||_2$ the L2 norm, and the mean projection error over the samples writes as

$$\boldsymbol{\sigma}_{POD} = \frac{1}{2N}\sum_{j=1}^{S} |(\boldsymbol{U})_j - (\boldsymbol{U}_{POD})_j| \in \mathbb{R}^H, \tag{4.10}$$

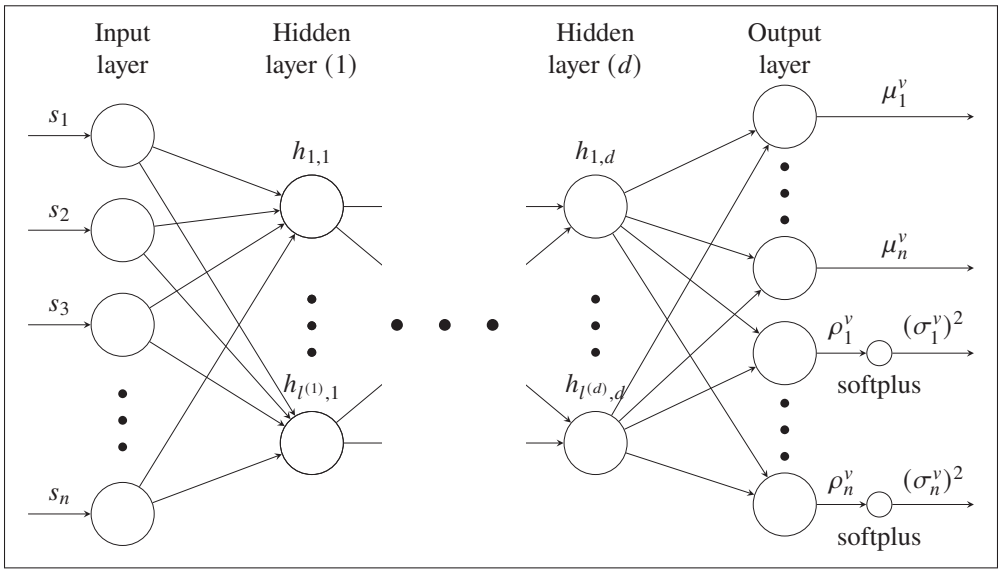


Figure 4.1 $\hat{u}_{DB}(\boldsymbol{X}; \boldsymbol{w}, \boldsymbol{b}) \sim \mathcal{N}(\boldsymbol{\mu}^v, (\boldsymbol{\sigma}^v)^2)$, a Deep Neural Network regression with a dual mean and variance output

4.3 Learning distributions over the expansion coefficients using Deep Ensembles

4.3.1 Regression objective

Building a non-intrusive ROM involves a statistical step to construct the function responsible for infering the expansion parameters $\boldsymbol{v}$ from new non-spatial parameters $\boldsymbol{s}$. This regression step is performed offline, and since we've considered the spatial parameters $\boldsymbol{x}$ to be externally handled, one can denote it as a mapping u_{DB} outputting the projection coefficients $\boldsymbol{v}(\boldsymbol{s})$, such as

$$\begin{aligned} u_{DB} : \mathbb{R}^P &\to \mathbb{R}^L \\ \boldsymbol{s} &\mapsto \boldsymbol{v}(\boldsymbol{s}). \end{aligned} \tag{4.11}$$

4.3.2 Deep Neural Networks with built-in variance

This statistical step is handled in the POD-NN framework by infering the mapping with a Deep Neural Network $\hat{u}_{DB}(\boldsymbol{s}; \boldsymbol{w}, \boldsymbol{b})$. $\boldsymbol{w}$ and $\boldsymbol{b}$ are the model parameters: the *weights* and *biases* of the network, and are learned during training (*offline* phase), to be later reused to make predictions

(*online* phase). Its number of hidden layers is called the *depth* d, which is chosen not to account for the input and output layers. Each layer has a number of neurons, called the *width* $l^{(j)}$.
The difference here with a vanilla DNN architecture for regression resides in the dual-output, first presented in Nix & Weigend (1994) and reused in Lakshminarayanan *et al.* (2017), where the final layer size is twice the number of expansion coefficients to project, $l^{(d+1)} = 2L$ since it both outputs a *mean* value μ and a *raw variance* ρ, which will then be constrained for positiveness through a softplus function, finally outputting σ^2 as

$$\sigma^2 = \log(1 + \exp(\rho)). \tag{4.12}$$

A representation of this DNN is pictured in Figure 4.1, with d hidden layers—and therefore, $d + 2$ layers in total. Each hidden layer state $\boldsymbol{h}^{(j)}$ gets computed from its input $\boldsymbol{h}^{(j-1)}$ alongside the layer weights $\boldsymbol{w}^{(j)}$ and biases $\boldsymbol{b}^{(j)}$, and finally goes through an activation function ϕ

$$\boldsymbol{h}^{(j)} = \phi\left(\boldsymbol{w}^{(j)}\boldsymbol{h}^{(j-1)} + \boldsymbol{b}^{(j)}\right), \tag{4.13}$$

with $\boldsymbol{h}^{(0)} = \boldsymbol{s}$, the input of $\hat{u}_{DB}$, and $\boldsymbol{h}^{(d+1)} = [\boldsymbol{\mu}^v,\ \boldsymbol{\rho}^v]^\intercal$, the output.
Since this predicted variance reports the spread, or noise, in data (the inputs data are drawn from a distribution), hence wouldn't be reduced even if we were to grow our dataset larger, it accounts for *aleatoric uncertainty*, which is usually separated from *epistemic uncertainty*, the one that is inherent to the model, Kendall & Gal (2017).
One can think about this idea of aleatoric uncertainty as a measurement problem: with the goal of measuring a quantity u, the tool has some inherent noise n, random and dependent on the parameter x in the measurable domain, making the measured quantity $u(x) + n(x)$. The model presented here, as introduced in Nix & Weigend (1994), is trying to perform the regression on both components, with an estimated variance alongside the regular point-estimate of the mean.

4.3.3 Ensemble training

Considering a N-sized training dataset $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$, with $\boldsymbol{X}$ denoting the normalized non-spatial parameters $\boldsymbol{s}$ and $\boldsymbol{v}$ the corresponding expansion coefficients coming from a training/validation-

split of the matrix of snapshots $\boldsymbol{U}$, an *optimizer* performs several *training epochs* N_e to minimize the following Negative Log-Likelihood loss function, w.r.t. the network weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$

$$\mathcal{L}_{\text{NLL}}(\mathcal{D}, \boldsymbol{\theta}) := \frac{1}{N} \sum_{i=1}^{N} \left[\frac{\log \ \boldsymbol{\sigma}_{\boldsymbol{\theta}}^{v\,2}(\boldsymbol{X})_i}{2} + \frac{(\boldsymbol{v} - \boldsymbol{\mu}_{\boldsymbol{\theta}}^{v}(\boldsymbol{X}))_i^2}{2\boldsymbol{\sigma}_{\boldsymbol{\theta}}^{v\,2}(\boldsymbol{X})_i} \right], \tag{4.14}$$

with the normalized inputs $\boldsymbol{X}$, as well as $\boldsymbol{\mu}_{\boldsymbol{\theta}}^{v}$ and $\boldsymbol{\sigma}_{\boldsymbol{\theta}}^{v\,2}$, respectively, the mean and variance retrieved from the network, parameterized by $\boldsymbol{\theta} = (\boldsymbol{w}, \boldsymbol{b})$.

This loss gets an additional term in practice, an L2 regularization, commonly known as *weight decay* in Neural Network contexts, Krogh & Hertz (1992), producing

$$\mathcal{L}_{\text{NLL}}^{\lambda}(\mathcal{D}, \boldsymbol{\theta}) := \mathcal{L}_{\text{NLL}}(\mathcal{D}, \boldsymbol{\theta}) + \lambda ||\boldsymbol{w}||^2. \tag{4.15}$$

Optimizers based on Stochastic Gradient Descent, such as Adam, Kingma & Ba (2014), are needed to handle this loss function, often irregular and non-convex in a Deep Learning context. The derivative of the loss $\mathcal{L}_{\text{NLL}}$ w.r.t. the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ is obtained through *automatic differentiation*, Rumelhart *et al.* (1986), a technique requiring to keep track of the gradients during the forward pass of the network, (4.13). Using *backpropagation*, Linnainmaa (1976), the updated weights $\boldsymbol{w}^{n+1}$ and biases $\boldsymbol{b}^{n+1}$ corresponding to the epoch $n+1$ write as

$$\left(\boldsymbol{w}^{n+1}, \boldsymbol{b}^{n+1}\right) = (\boldsymbol{w}^n, \boldsymbol{b}^n) - \eta f \left(\frac{\partial \mathcal{L}(\boldsymbol{w}^n, \boldsymbol{b}^n; \boldsymbol{X}, \boldsymbol{v})}{\partial (\boldsymbol{w}^n, \boldsymbol{b}^n)} \right), \tag{4.16}$$

with $f(\cdot)$ a function of loss derivative w.r.t. weights and biases that depends on the optimizer choice, and η the *learning rate*, a hyperparameter defining the step size taken by the optimizer. The idea behind Deep Ensembles, presented in Lakshminarayanan *et al.* (2017) and recommended in Snoek *et al.* (2019), is to randomly initialize M sets of $\boldsymbol{\theta}_m = (\boldsymbol{w}, \boldsymbol{b})$, therefore creating M independent NNs. Each of them is then subsequently trained. Overall, the predictions moments in the reduced space $(\boldsymbol{\mu}_{\boldsymbol{\theta}_m}^{v}, \boldsymbol{\sigma}_{\boldsymbol{\theta}_m}^{v})$ of each create a probability mixture, that, as suggested by the original authors, we approximate in a single Gaussian distribution, leading to a mean expressed

as

$$\boldsymbol{\mu}_*^v = \frac{1}{M}\sum_{m=1}^{M}\boldsymbol{\mu}_{\boldsymbol{\theta}_m}^v, \tag{4.17}$$

and a variance subsequently obtained as

$$\sigma_*^{v2} = \frac{1}{M}\sum_{m=1}^{M}\left[(\sigma_{\boldsymbol{\theta}_m}^v)^2 + (\mu_{\boldsymbol{\theta}_m}^v)^2\right] - \mu_*^2. \tag{4.18}$$

The model is now accounting for the *epistemic uncertainty* through random initialization and variability in the training step. This uncertainty is directly linked to the model, and could be reduced, for we had more data to feed it with. It is directly related to the data-fitting capabilities of the model, and will, therefore, snowball in the absence of such data, since there is no more constraint. It has in our case the most value, compared to aleatoric uncertainty, since one of our objectives is being warned when the model is making predictions out-of-distribution.

Since these networks are independent, parallelizing their training is relatively easy, with only the results having to be averaged over. We will refer to this model as POD-EnsNN.

Algorithm 4.1 Deep Ensembles training and predictions

Prepare the dataset $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$
for *each model in the ensemble* $1 \leq m \leq M$ **do**
    Train the model m:
    for *each epoch* $1 \leq e \leq N_e$ **do**
        Retrieve the outputs $(\boldsymbol{\mu}_{\boldsymbol{\theta}_m}^v, \boldsymbol{\rho}_{\boldsymbol{\theta}_m}^v)$ from the forward pass $\hat{u}_D(\boldsymbol{X})$
        Perform the variance treatment, $\boldsymbol{\sigma}_{\boldsymbol{\theta}_m}^{v^2} = \text{softplus}(\boldsymbol{\rho}_{\boldsymbol{\theta}_m}^v)$
        Compute the loss $\mathcal{L}_{\text{NLL}}$
        Backpropagate the gradients to the parameters $\boldsymbol{\theta}_m$
    end
    Retrieve statistical outputs $(\boldsymbol{\mu}_{\boldsymbol{\theta}_m}^v, (\boldsymbol{\sigma}_{\boldsymbol{\theta}_m}^v)^2)$ for the model m for a test dataset
end
Approximate the predictions for the reduced space in a Gaussian $\mathcal{N}(\boldsymbol{\mu}_*^v, \boldsymbol{\sigma}_*^{v2})$

4.3.4 Metrics

In addition to the regularized loss $\mathcal{L}^{\lambda}_{\text{NLL}}$, we define a relative error RE that is more meaningful to report the results with

$$RE(\hat{\boldsymbol{U}}, \boldsymbol{U}) = \frac{|| \sum_{i=1}^{S} (\hat{\boldsymbol{U}}_i - \boldsymbol{U}_i) ||_2}{|| \sum_{i=1}^{S} \boldsymbol{U}_i ||_2}, \tag{4.19}$$

with $\boldsymbol{U} = \boldsymbol{V}.\boldsymbol{v}$ and $\hat{\boldsymbol{U}} = \boldsymbol{V}.\hat{\boldsymbol{v}}$. In both cases, $\boldsymbol{V}$ corresponds to the POD bases, extracted in Section 4.2. During the training, we report two metrics: the training loss $\mathcal{L}^{\lambda}_{\text{NLL}}$, as well as the validation relative error RE_{val}.

4.3.5 Predictions in the expanded space

While embedding uncertainty quantifications within Deep Neural Networks helps getting a confidence interval on the predicted expansion coefficients $\boldsymbol{v}$, there is still a need to perform the expansion step subsequently to retrieve the full solution, and it is defined as a dot product with the modes matrix $\boldsymbol{V}$, as defined in (4.7).

While this applies perfectly for the predicted mean μ^v, one needs to be careful when handling the predicted standard deviation σ^v since there is no theoretical guarantee that the statistical moments on the reduced basis translate linearly to the expanded space. However, the distribution over the coefficients $\boldsymbol{v}$ is known as following

$$\hat{\boldsymbol{v}} = \hat{u}_{DB}(\boldsymbol{X}; \boldsymbol{w}, \boldsymbol{b}) \sim \mathcal{N}\left(\boldsymbol{\mu}^v_*, \boldsymbol{\sigma}^{v\,2}_*\right). \tag{4.20}$$

Therefore, unlimited samples $\hat{\boldsymbol{v}}^{(i)}$ can be drawn from it, and individually decompressed into a corresponding full solution $\hat{u}^{(i)}_D = \boldsymbol{V}.\hat{\boldsymbol{v}}^{(i)}$, from (4.7). We hence put forward the following Monte-Carlo approximation of the full distribution on $\hat{u}_D$, drawing N_{ex} samples, and computing

$$\boldsymbol{\mu}_* = \frac{1}{N_{\text{ex}}} \sum_{i=1}^{N_{\text{ex}}} \hat{u}^{(i)}_D = \frac{1}{N_{\text{ex}}} \sum_{i=1}^{N_{\text{ex}}} \boldsymbol{V}.\boldsymbol{v}^{(i)}, \tag{4.21}$$

$$\boldsymbol{\sigma}^2_* = \frac{1}{N_{\text{ex}}} \sum_{i=1}^{N_{\text{ex}}} \left[\hat{u}^{(i)}_D - \boldsymbol{\mu}_*\right]^2 = \frac{1}{N_{\text{ex}}} \sum_{i=1}^{N_{\text{ex}}} \left[\boldsymbol{V}.\boldsymbol{v}^{(i)} - \boldsymbol{\mu}_*\right]^2, \tag{4.22}$$

which represent the approximated statistical moments of the distribution on the predicted full solution $\hat{u}_D(\boldsymbol{s})$, also refered to as $\hat{u}_D^{\mu}$ and $\hat{u}_D^{\sigma}$.

4.3.6 Adding adversarial training

First proposed in Szegedy, Zaremba, Sutskever, Bruna, Erhan, Goodfellow & Fergus (2014) and studied in Goodfellow, Shlens & Szegedy (2014b), the concept of *adversarial training*, not to be confused with Generative Adversarial Networks, Goodfellow *et al.* (2014a), aims at improving the robustness of neural networks when confronted with noisy data, that could potentially be intentionally created.

In the Deep Ensembles framework, it is an optional component that, according to Lakshminarayanan *et al.* (2017), can help to smooth the output out, and could be useful as will be shown in the oncoming test case, where the POD is struggling with the highly-nonlinear wave getting produced by Burgers' equation, see Section 4.4.2.

A simple implementation is the *gradient sign* technique, which is adding noise in the opposite way of the gradient descent, scaled by a new hyperparameter ζ, at each training epoch, and is pictured in Algorithm 4.2. The idea is indeed to perform *data augmentation* at each training epoch. The additional data comes from the generated adversarial samples which will help train the network in a more robust way since these tricky samples are being inserted in the dataset.

Algorithm 4.2 Implementing adversarial training within the training loop

Function `getAdversarialLoss(`$\boldsymbol{X}$, $\boldsymbol{v}$, ϵ`)`**:**
  $\mathcal{L}_T \leftarrow \mathcal{L}(\{\hat{u}_D(\boldsymbol{X}), \boldsymbol{v}\}, \boldsymbol{\theta})$
  $\boldsymbol{X}' \leftarrow \boldsymbol{X} + \zeta \ \text{sign}(\frac{\mathcal{L}_T}{\partial \boldsymbol{X}})$
  $\mathcal{L}_T \leftarrow \mathcal{L}_T + \mathcal{L}(\{\hat{u}_D(\boldsymbol{X}'), \boldsymbol{v}\}, \boldsymbol{\theta})$
return $\mathcal{L}_T$

4.4 Benchmarks with uncertainty quantification

In this section, we assess the uncertainty propagation component of our framework against two benchmark problems, using the same setup each time. The first is steady and two-dimensional,

known as the Ackley Function, while the second involves a solution of Burgers' equation, which is time-dependent and one-dimensional.

The library TensorFlow, Abadi (2016), in version 2.1.0 is used for all results, while the SVD algorithm and various matrix operations are performed by NumPy, all in Python 3.7. Documented source code will be made available at https://github.com/pierremtb/POD-UQNN, on the `POD-EnsNN` branch.

In both benchmarks, the *activation function* on all hidden layers is the ReLU nonlinearity $\phi : x \mapsto \max(0, x)$, while a linear mapping is applied on the output layer since in a regression case, real-valued variables are needed as outputs. We choose the NN topology to involve $d = 3$ hidden layers, of widths $l^{(1)} = l^{(2)} = l^{(3)} = 128$. We perform normalization on all non-spatial parameters $\boldsymbol{s}$, to build the inputs $\boldsymbol{X}$ as

$$\boldsymbol{X} = \frac{\boldsymbol{s} - \bar{\boldsymbol{s}}}{\boldsymbol{s}_{\text{std}}}, \tag{4.23}$$

with $\bar{\boldsymbol{s}}$ and $\boldsymbol{s}_{\text{std}}$ being respectively the empirical mean and standard deviation over the dataset, on each column to keep physical meaning, e.g., the time would be normalized w.r.t. time moments.

To achieve GPU parallel training, we used the Horovod library, Sergeev & Del Balso (2018), which allowed us to efficiently train the $M = 5$ models on $M = 5$ GPUs at the same time, this number being the recommended one from Lakshminarayanan *et al.* (2017).

Among the following benchmarks, the Burgers' equation solution is time-dependent, which grows the matrix of snapshots $\boldsymbol{U}$ substantially. We, therefore, make use of the two-step POD algorithm, presented in Wang *et al.* (2019). The results are comforting: on a dataset of size $S = 10,000$, with $N_t = 100$, the time to compute the SVD decomposition shrunk from 0.63 seconds to 0.51 switching from using the regular POD to the two-step POD algorithm, which could result in a significant gain on more massive datasets. Numba optimizations have also been used for both the regular POD algorithm and dual one, as well as for data generation, which allows for multi-threading and native code compilation within Python, Lam *et al.* (2015), and is especially useful for loop-based computations.

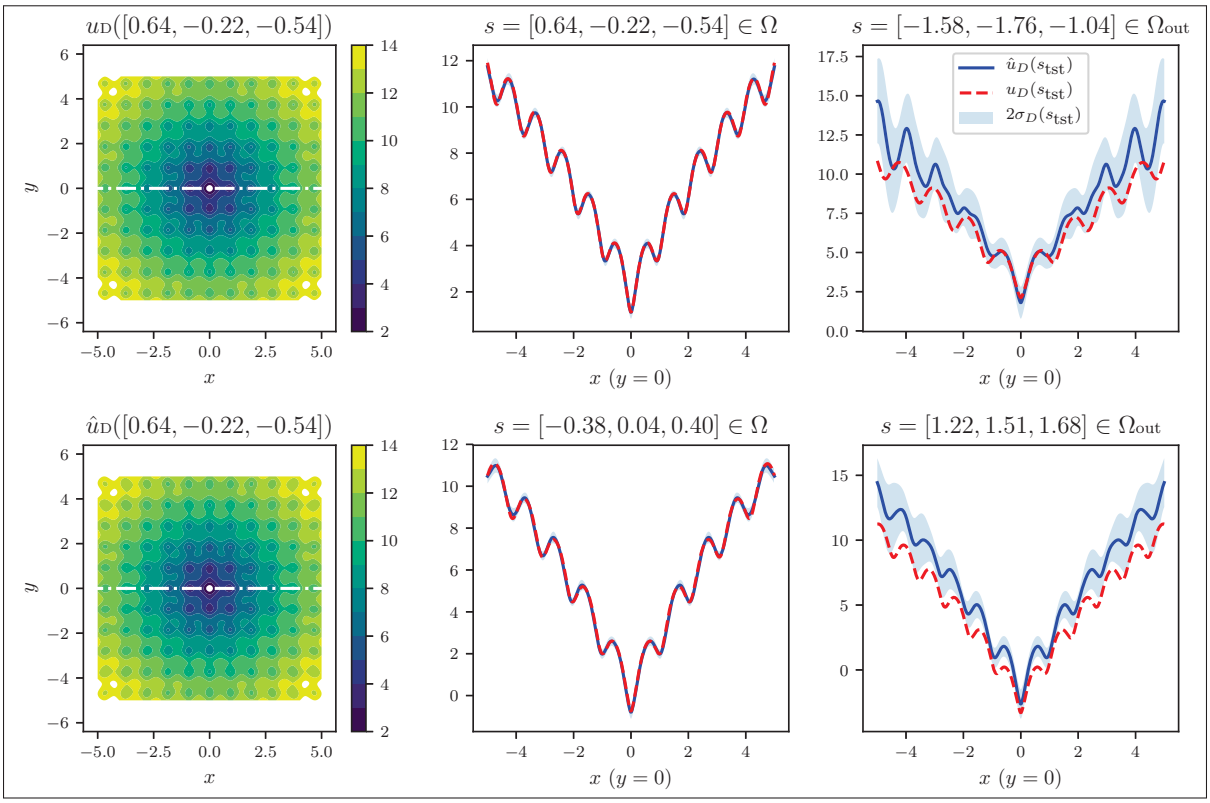


Figure 4.2 Ackley Function (2D). The first column is a quick visualization to see contour plots of the predicted mean over the testing samples $u_D(s_{\text{tst}})$ on top, and the true mean at the bottom. The second column shows the predicted $\hat{u}_D$ and the observed data u_D from the dataset across two random snapshots inside the training bounds, within the test set (top/bottom). The third column shows results for the samples s_{out}, that are taken outside the dataset bounds and have therefore more substantial uncertainties.

4.4.1 Stochastic Ackley function

As a first test case, we introduce a stochastic version of the Ackley function, a highly irregular baseline with multiple extrema presented in Sun *et al.* (2019), which takes $P = 3$ parameters.

Being real-valued ($D = 1$) and two-dimensional in space ($n = 2$), it is defined as

$$
\begin{aligned}
u : \mathbb{R}^{2+P} &\to \mathbb{R} \\
(x, y; \boldsymbol{s}) &\mapsto -20\,(1 + 0.1 s_3) \exp\left(-0.2(1 + 0.1 s_2)\sqrt{0.5(x^2 + y^2)}\right) \\
&\quad - \exp\left(0.5(\cos(2\pi(1 + 0.1 s_1)x) + \cos(2\pi(1 + 0.1 s_1)y))\right) \\
&\quad + 20 + \exp(0),
\end{aligned}
\tag{4.24}
$$

with the non-spatial parameters vector $\boldsymbol{s}$ of size $P = 3$, each element s_i randomly sampled over $\Omega_{s_i} = [-1, 1]$, as performed by authors of Sun *et al.* (2019).

The 2D space domain $\Omega_{xy} = [-5, 5] \times [-5, 5]$ is discretized linearly in $N_{x_1} = N_x = 400$ and $N_{x_2} = N_y = 400$, leading the number of DOFs to be $H = 160,000$.

With $S = N_S = 500$ as our default number of samples of the parameters $\boldsymbol{s}$, and we use a Latin Hypercube Sampling (LHS) strategy to sample each non-spatial parameter on their domain $\Omega = [-1, 1]$ and generate the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times S}$.

After getting the reduced POD bases via (4.3–4.7) picking $\epsilon = 10^{-4}$, producing $L = 79$ coefficients to be matched by half of the final layer, and generating a full set of inputs and targets, the $(80\%, 20\%)$ *train*/*validation* ratio splits it into $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$ of size $N = 400$, $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{v}_{\text{val}}\}$ of size $N_{\text{val}} = 100$. We generate an additional test set $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{v}_{\text{tst}}\}$ of size $N_{\text{tst}} = 300$.

A fixed learning rate of $\eta = 0.005$ is set for the Adam optimizer, as well as an L2 regularization with the coefficient $\lambda = 10^{-3}$. No mini-batch split is performed since our dataset is small enough to be fully handled in memory. The training epochs count is $N_e = 25,000$.

The training of each model took 37 seconds, 37 seconds, 38 seconds, 38 seconds, and 38 seconds on each GPU, and the total, real-time of the parallel process was 1 minute and 2 seconds. In order to picture the random initialization of each model, here are the training losses: $\mathcal{L} = 4.0548 \times 10^0$, 4.5826×10^0, 4.8950×10^0, 4.8916×10^0, and 3.9446×10^0, down from the initial $\mathcal{L}_0 = 2.7332 \times 10^6$, 3.1626×10^6, 2.9548×10^6, 2.8836×10^6, and 2.9711×10^6.

The overall relative errors reached were $RE_{\text{val}} = 1.12\%$ and $RE_{\text{tst}} = 1.11\%$, for validation and testing, respectively.

The first column of Figure 4.2 shows two contour plots of the predicted mean across the testing set as well as the true mean, to quickly visualize the Ackley function, its irregularity and, its various local extrema. The second column shows two different random samples within the same testing set with predictions and analytical values, while the third column contains *out-of-distribution* cases, sampled in $\Omega_{s_{i,\text{out}}}$, defined as

$$\Omega_{s_{i,\text{out}}} = [-2, -1] \cup [1, 2]. \tag{4.25}$$

The essential thing to notice in this last column of Figure 4.2 is the two slices of parameters that are sampled *out-of-distribution*, meaning that are outside of the dataset bounds. We can see that the predicted mean, represented by the continuous blue line, is performing badly w.r.t. the red dashed line, which represents the true values. This predicted mean would be approximately the same as the point estimate prediction of a regular Deep Neural Network. And even if here, our mean prediction out-of-distribution is indeed off, from the wide confidence zone defined by the two standard deviations on the prediction, we get a warning that the model *doesn't know*, and therefore doesn't try to make a precise claim.

4.4.2 Burgers' equation solution

Our second test case is very different, chosen to asses the flexibility of the framework properly. It's a solution of the viscous Burgers' equation, that is notoriously hard to work with for computational methods because of its shock-forming behavior, Raissi *et al.* (2019a). It can take in our case $P = 1$ stochastic parameter, the fluid viscosity, denoted here as s. Being real-valued ($D = 1$) and one-dimensional in space, yet time-dependent, it is defined as

$$\begin{aligned} u : \mathbb{R}^{2+1} &\to \mathbb{R} \\ (x, t; s) &\mapsto \tilde{u}(x, t; s), \end{aligned} \tag{4.26}$$

with the non-spatial parameters vector $\boldsymbol{s} = s$ of size $P = 1$, and $\tilde{u}(x, t; s)$ being an analytically available solution of the following PDE definition, which is a case of Burgers' equation with an initial sine condition, as presented in Basdevant *et al.* (1986). The subscripts are denoting the

partial derivatives, defining it as

$$u_t + uu_x - su_{xx} = 0, \quad x \in \Omega_x = [0, 1.5], \quad t \in \Omega_t = [1, 5], \tag{4.27}$$

$$\begin{aligned} u(0,t) = u(1.5,t) &= 0, \quad 1 \leq t, \\ u(x,1) &= \frac{x}{1 + \exp\left[\frac{1}{4s}(x^2 - \frac{1}{4})\right]}, \quad 0 < x < 1.5. \end{aligned} \tag{4.28}$$

There exists a directly available analytical solution according to Maleewong & Sirisup (2011), expressed as

$$\tilde{u}(x,t,s) = \frac{x/t}{1 + (t/t_0)^{1/2} \exp\left(\frac{x^2}{4st}\right)}, \quad 1 \leq t, \tag{4.29}$$

with $t_0 = \exp(1/8s)$.

The 1D space domain $\Omega_x = [0, 1.5]$ is discretized linearly in $N_x = 256$, and since it's real-valued, the number of DOFs remains $H = 256$.

With $N_t = 100$ time-steps in the domain $\Omega_t = [1, 5]$, we generate $N_s = 100$ samples of the parameters $\boldsymbol{s}$ using an LHS strategy over the domain $\Omega_s = [0.001, 0.010]$ and produce the matrix of snapshots $\boldsymbol{U} \in \mathbb{R}^{H \times N_s N_t}$. This domain is chosen to be spread around the value of $s = 0.005$ used in Maleewong & Sirisup (2011) and similar to $s = 0.01/\pi$ in Raissi *et al.* (2019a).

After getting the reduced POD bases via (4.3–4.7) and the dual-step POD approach, picking $\epsilon = 10^{-10}$ and $\epsilon_0 = 10^{-8}$, producing $L = 20$ coefficients to be matched by half of the final layer, and generating a full set of inputs and targets, we use our default $(80\%, 20\%)$ *train*/*validation* ratio to split it into $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$ of size $N = 80 * 100$, $\mathcal{D}_{\text{val}} = \{\boldsymbol{X}_{\text{val}}, \boldsymbol{v}_{\text{val}}\}$ of size $N_{\text{val}} = 20 * 100$. We produce an additional test set $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{v}_{\text{tst}}\}$ of size $N_{\text{tst}} = 100 * 100$.

A fixed learning rate of $\eta = 0.01$ is set for the Adam optimizer, as well as an L2 regularization with the coefficient $\lambda = 10^{-8}$. No mini-batch split is performed since our dataset remains small enough to be fully handled in memory, even though the time dimension is considerably increasing the total size.

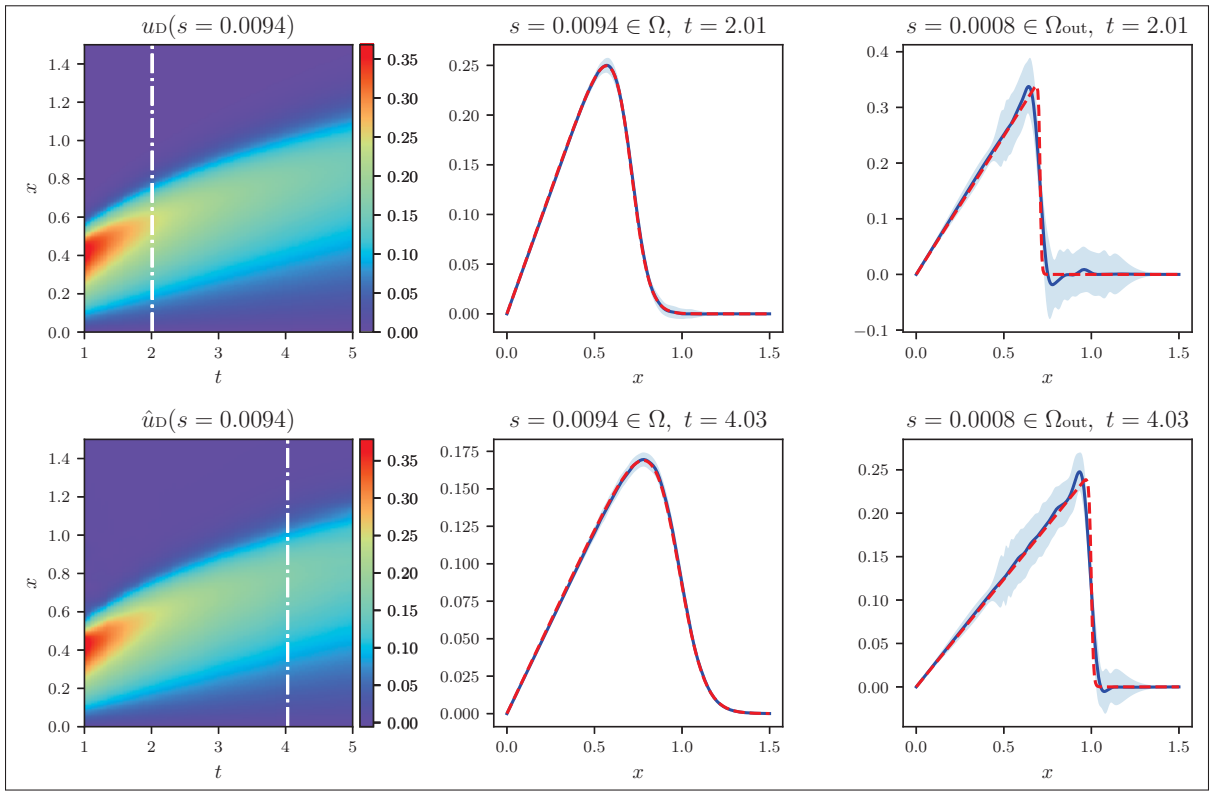


Figure 4.3 Burgers' equation (1D, unsteady). As a quick visualization, one can see colormaps of a random test sample of the first column, as well as the time-steps depicted by the white lines. Then, from left to right: comparing the predicted $\hat{u}_D$ and the analytical data u_D from the dataset at the time-steps, and across two random snapshots for the viscosity parameter s, respectively in and out of the training bounds, and one can see the uncertainty increasing while exiting the training bounds

We choose a fixed learning rate of $\eta = 0.01$ for the Adam optimizer, as well as an L2 regularization with $\lambda = 10^{-8}$. No mini-batch split is performed. The training epochs number is set to $N_e = 13,000$.

The training of each model took 52, 52, 52, 52, and 52 seconds on each GPU, and the total, real-time of the parallel process was 1 minute and 15 seconds. In order to picture the random initialization of each model, here are the training losses: $\mathcal{L} = -4.8198 \times 10^0$, -4.3091×10^0, -5.1255×10^0, -5.0622×10^0, and -5.0182×10^0, down from the initial $\mathcal{L}_0 = -4.7256 \times 10^{-2}$, -4.3737×10^{-2}, -2.9728×10^{-2}, -4.3704×10^{-2}, and -5.5732×10^{-2}.

The overall relative errors reached were $RE_{\text{val}} = 1.33\%$ and $RE_{\text{tst}} = 1.17\%$, for validation and testing, respectively.

Figure 4.3 shows on its first column colormaps for a random test sample, with on top the analytical solution and on the bottom the predicted solution. Then, on the second, one can see great performances for the test predictions on the same sample at two different time-steps, which were depicted as white lines on the first column. Finally, the last column is meant for out-of-distribution predictions, with a sample from the domain Ω_{out}, defined as

$$\Omega_{\text{out}} = [0.0005, 0.001] \cup [0.010, 0.0105]. \tag{4.30}$$

Again, this last column shows the ability of the ensembles-enhanced POD-NN framework to show a warning when one aims for the outside of the dataset bounds, with larger confidence zones, and therefore intentionally less precise predictions.

4.5 Flood Modeling application: the Mille-Iles river

After assessing how the Deep Ensembles version of the POD-NN model performed on various benchmark problems with different dimensions and time-dependencies in Section 4.4, this one will aim at applying it to real-world engineering problems: flood modeling.

4.5.1 Background

Just like wildfires or hurricanes, floods are natural phenomena that can be devastating, especially in densely populated areas. Around the globe, they have become more and more frequent, and ways to predict it should be found to deploy safety services and evacuate areas when needed.

The primary physical phenomenon in flooding predictions involves *free surface flows* and is usually described by the Shallow Water equations for rivers and lakes, extensively described in Toro (2001), which, in their inviscid form, are defined as follows

$$\frac{\partial}{\partial t}\int_{\Omega_{xy}} \boldsymbol{U}\, d\Omega_{xy} + \int_{\partial\Omega_{xy}} ([\boldsymbol{G}(\boldsymbol{U})\ \boldsymbol{H}(\boldsymbol{U})] \cdot \boldsymbol{n})\, d\Gamma = \int_{\Omega_{xy}} \boldsymbol{S}(\boldsymbol{U})\, d\Omega_{xy} \quad \text{on } [0, T_s], \tag{4.31}$$

with T_s denoting the time duration, and

$$\boldsymbol{U} = \begin{bmatrix} h \\ hv_x \\ hv_y \end{bmatrix}, \quad \boldsymbol{G}(\boldsymbol{U}) = \begin{bmatrix} hv_x \\ hv_x^2 + \frac{1}{2}gh^2 \\ hv_x v_y \end{bmatrix}, \quad \boldsymbol{H}(\boldsymbol{U}) = \begin{bmatrix} hv_x \\ hv_y^2 + \frac{1}{2}gh^2 \\ hv_x v_y \end{bmatrix},$$

$$\boldsymbol{S}(\boldsymbol{U}) = \begin{bmatrix} 0 \\ gh(S_{0_x} - S_{f_x}) \\ gh(S_{0_x} - S_{f_y}) \end{bmatrix}, \quad \begin{bmatrix} S_{0_x} \\ S_{0_y} \end{bmatrix} = -\nabla z,$$

$$\text{and } \boldsymbol{S}_f = \begin{bmatrix} S_{f_x} \\ S_{f_y} \end{bmatrix} = \begin{bmatrix} \dfrac{m^2 v_x \sqrt{v_x^2 + v_y^2}}{h^{4/3}} \\ \dfrac{m^2 v_y \sqrt{v_x^2 + v_y^2}}{h^{4/3}} \end{bmatrix},$$

considering the h water depth, (v_x, v_y) the velocity components, m the Manning roughness, g the gravity density, $\boldsymbol{S}_f$ the friction vector, and z the bottom depth w.r.t. a reference level.
These equations can be discretized using finite volumes, as detailed in Toro (2001) and Zokagoa & Soulaimani (2012b). And while we do already have decent numerical simulation programs to make these predictions, with well-validated software like *TELEMAC*, Galland, Goutal & Hervouet (1991) or *CuteFlow*, Zokagoa & Soulaimani (2012b), these are both computational and time-expensive for multi query simulations as in uncertainties propagation. Therefore, it is impossible to run them in real-time since they depend on various stochastic parameters. The POD-NN model, enriched with uncertainty quantification via Deep Ensembles, aims precisely at addressing this type of issue.

4.5.2 A 1D test case

We first put forward a one-dimensional test case in the Shallow Water equations application, with two goals in mind. The first is to have a reproducible benchmark on the same equations that will be used for flood modeling, with an analytically available solution, and, therefore, generable

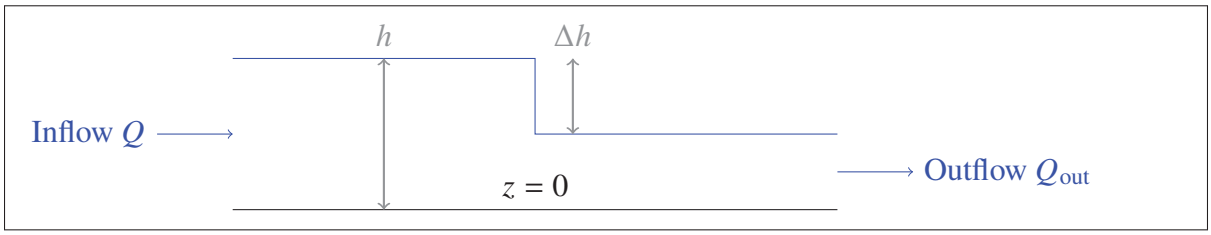


Figure 4.4 Simple representation of the water flow and main quantities before a dam break ($\Delta h > 0$)

data. The second is to make sure that the solver CuteFlow performs correctly w.r.t. the analytical solution, since in future sections, it will be our only data source.

The 1D domain $\Omega_{xy} = [0, 100]$ m is considered, with $N_x = 132$ points, uniformly distributed. An initial condition is considered, with two levels of water depth, $s = \Delta h$ denoting the difference, that will act as our stochastic parameter in this study, with the water depth in the outflow fixed at $h = 1$ m. Following the initial discontinuity at $t = 0$, we consider $N_t = 50$ time-steps, separated by $\Delta t = 0.1$ s, in the domain $\Omega_t = [0, 5]$ s. There are $N_D = 2$ DOFs per node, the water depth h and the velocity u, leading to the total number of DOFs $H = 264$.

The dataset for the training/validation $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$ of size $N = 40$, split with the usual 80%/20% ratio, is generated from an analytical solution sampling s in $\Omega = [2, 20]$, presented in Wu, Huang & Zheng (1999), as well as a testing dataset $\mathcal{D}_{\text{tst}} = \{\boldsymbol{X}_{\text{tst}}, \boldsymbol{v}_{tst}\}$ of size $N_t st$, with $\boldsymbol{s}_{\text{tst}} = [2, 3, \ldots, 20]^\intercal$ m. Additionally, the numerical finite volume solver CuteFlow is used to generate corresponding test solutions, from which we also obtain $N_t = 50$ time-steps after the initial condition, with a 2D dedicated mesh of 25551 nodes and 50000 elements specifically designed to represent this 1D problem in a compatible way for the solver.

The Python and TensorFlow implementation involves a topology three layers of 256 neurons for each network of the ensemble to encompass for nonlinearities. The POD handles the water depth h, and its truncation is performed with $\epsilon = 1.10^{-5}$, producing $L = 79$ coefficients to be matched by half of the final layer. $N_e = 30,000$ epochs are set for training. L2 regularization is used with a coefficient of $\lambda = 1.10^{-4}$, while adversarial training is set to $\zeta = 0.001$.

The training of each model took 49, 50, 50, 51, and 51 seconds on each GPU, and the total, real-time of the parallel process was 1 minute and 15 seconds, and reached the following losses

$\mathcal{L} = -1.0993 \times 10^0$, -1.0075×10^0, -1.0105×10^0, -1.0692×10^0, and -8.4801×10^{-1}, down from the initial $\mathcal{L}_0 = 1.3699 \times 10^2$, 1.3085×10^2, 1.3238×10^2, 1.3352×10^2, and 1.3545×10^2, reported to show the variance within the ensemble due to the random initialization.

The overall relative errors reached were $RE_{\text{val}} = 4.46\%$ and $RE_{\text{tst}} = 4.42\%$, for validation and testing, respectively.

The results are displayed in Figure 4.5 for the water depth, and Figure 4.6 for the velocity. On both, one can see two samples, with one within the testing set, that is pictured on the first column as a colormap for a visual purpose, and plotted for two time-steps on the second column. The first time-step is notably the initial condition, which is very well handled by the POD compression-expansion. One can note the black line in the second column, representing the corresponding solution computed by the numerical solver CuteFlow, that is very close to the analytical solution, and, therefore, validates it for later use in more complex cases.

A second out-of-distribution sample from $\Omega_{\text{out}} = [20, 30]$ m is plotted for the same two time-steps on the third column. The model performance within the training set is very decent considering the nonlinearities involved, with relatively small uncertainties, while it decreases when going out-of-distribution, as one expects it.

4.5.3 River setup

Our domain Ω_{xy} is composed of an unstructured mesh of $N_{xy} = 24361$ nodes, connected in 481930 triangular elements. It is represented in Figure 4.7. Each node has $N_{\text{val}} = 1$ degree of freedom, the water depth h on which the POD will be performed, leading to the global number of DOFs to be $H = 24361$. It is represented in Figure 4.8.

For this first study, we will consider the time-independent case, and we have at our disposal a dataset of $S = 180$ samples for different inflow discharge (Q) values that we will use for training, and one of $S = 20$ that we will use for testing, with the solution computed numerically with the software CuteFlow. They have both been randomly sampled before splitting in the domain $\Omega = [800, 1200]$ m^3/s. It is chosen to be mainly above the regular flow in the river of $Q_0 = 780$ m^3/s, Zokagoa & Soulaimani (2018).

We picked a POD truncating criterion of $\epsilon = 10.10^{-10}$, producing $L = 81$ coefficients to be matched by half of the final layer, a fixed learning rate of $\eta = 0.003$ for the Adam optimizer,

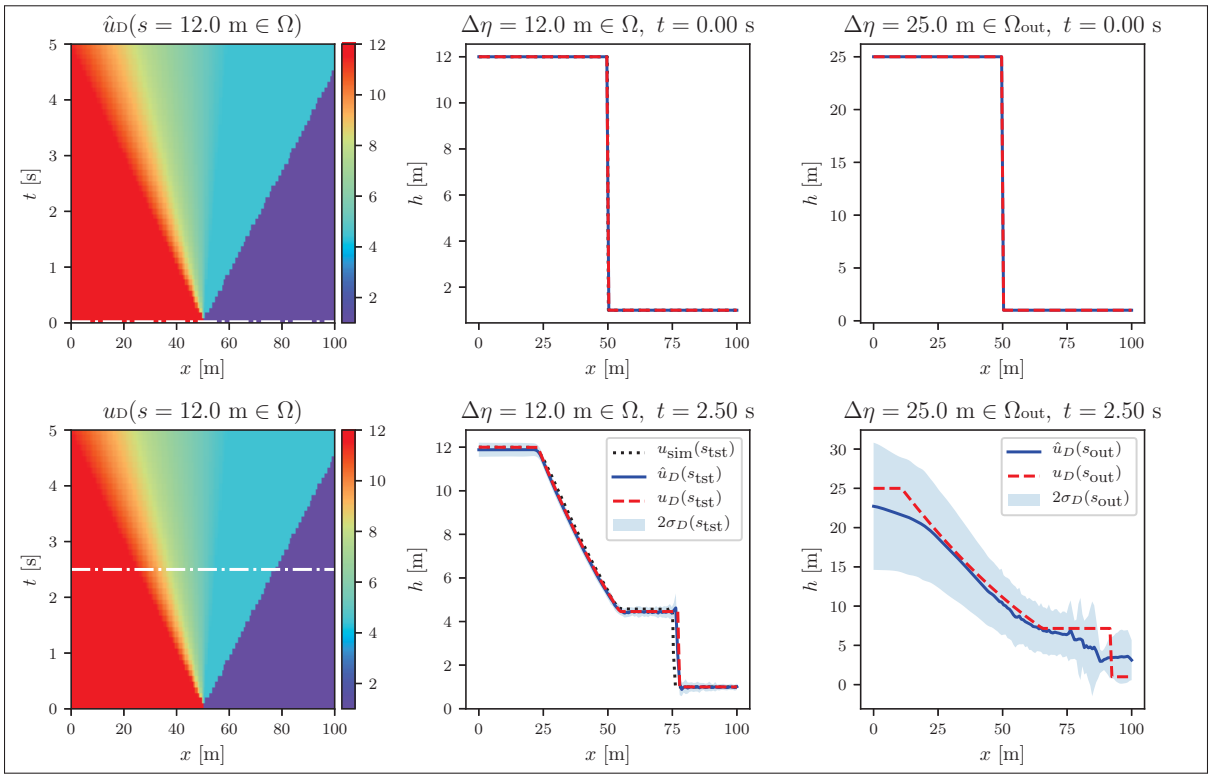


Figure 4.5 1D test case for SWE, water elevation results. The first two columns show results for a random sample in the test set, while the last column shows a random sample taken out-of-distribution. The white lines on the color maps denote the time steps of the last two columns. The lines u_{sim} are computed numerically by CuteFlow, and compared to the predicted mean $\hat{u}_D$ as well as the analytical value u_D

without L2 regularization, as well as the default Deep Ensembles activation function, ReLU. No mini-batching is performed, i.e., the whole dataset is run through at once for each epoch. The training epochs number is set to $N_e = 120,000$.

4.5.4 Results

The training of each model took 4 minutes 19 seconds, 4 minutes 20 seconds, 4 minutes 20 seconds, 4 minutes 21 seconds, and 4 minutes 21 seconds on each GPU, and the total, real-time of the parallel process was 4 minutes and 45 seconds. Again, to show the diversity in the five models, here are the final training losses: $\mathcal{L} = -2.2240 \times 10^0$, -3.5421×10^0, -3.5199×10^0, -2.3894×10^0, and -3.5003×10^0, down from the initial $\mathcal{L}_0 = 2.6522 \times 10^4$, 2.6075×10^4, 2.6522×10^4, 2.5357×10^4, and 2.6958×10^4.

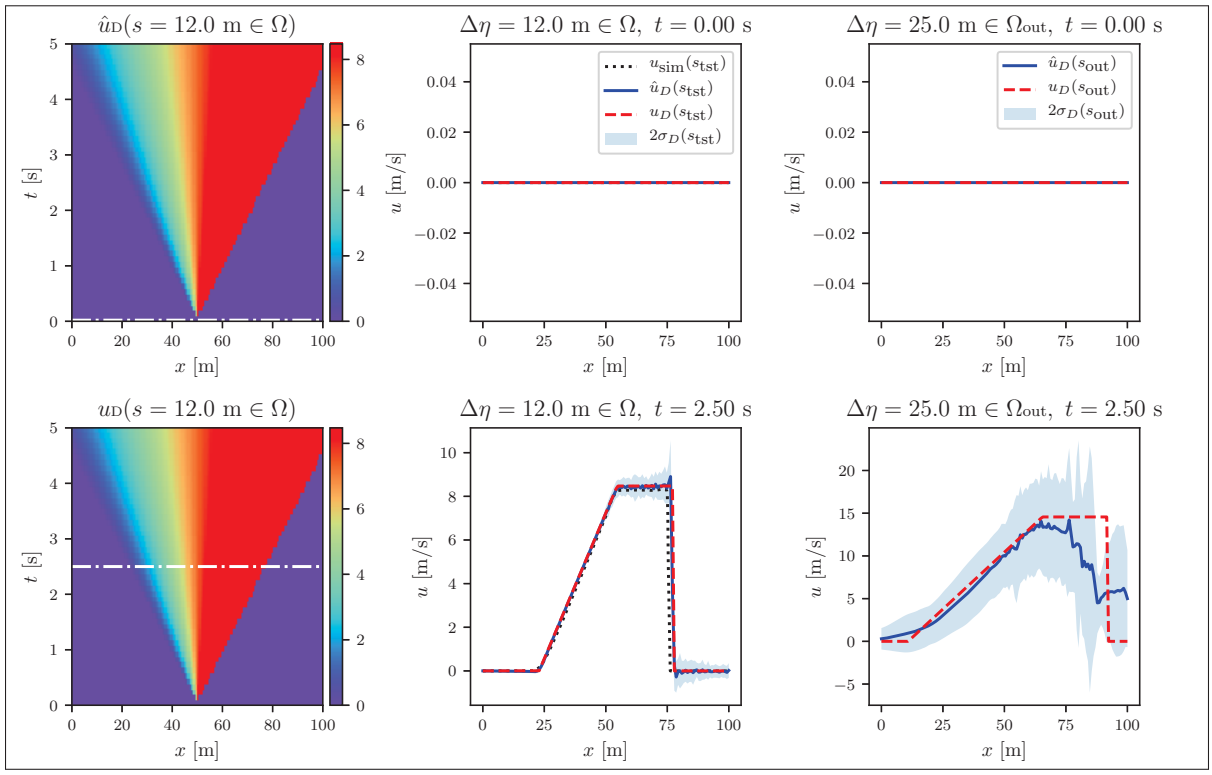


Figure 4.6 1D test case for SWE, velocity results. The first two columns show results for a random sample in the test set, while the last column shows a random sample taken out-of-distribution. The white lines on the color maps denote the time steps of the last two columns. The lines u_{sim} are computed numerically by CuteFlow, and compared to the predicted mean $\hat{u}_D$ as well as the analytical value u_D

The overall relative errors reached were $RE_{\text{val}} = 1.90\%$ and $RE_{\text{tst}} = 1.46\%$, for validation and testing, respectively.

Figure 4.8 depicts random test predictions using the open-source visualization software Paraview, Ahrens, Geveci & Law (2005), on two random samples for the water depth h. We notice the flooding limit, achieved by slicing at $h = 0.05\ m$ of water depth—in place of 0 for stability, are very well predicted when compared to the simulation results from CuteFlow (red line), and one can retrieve these additional white lines, adding ±2 standard deviations on top of the mean predictions, depicted by the blue body of water, that would define the confidence interval of the predicted flood lines. We consider that having this probability-distribution outcome, over

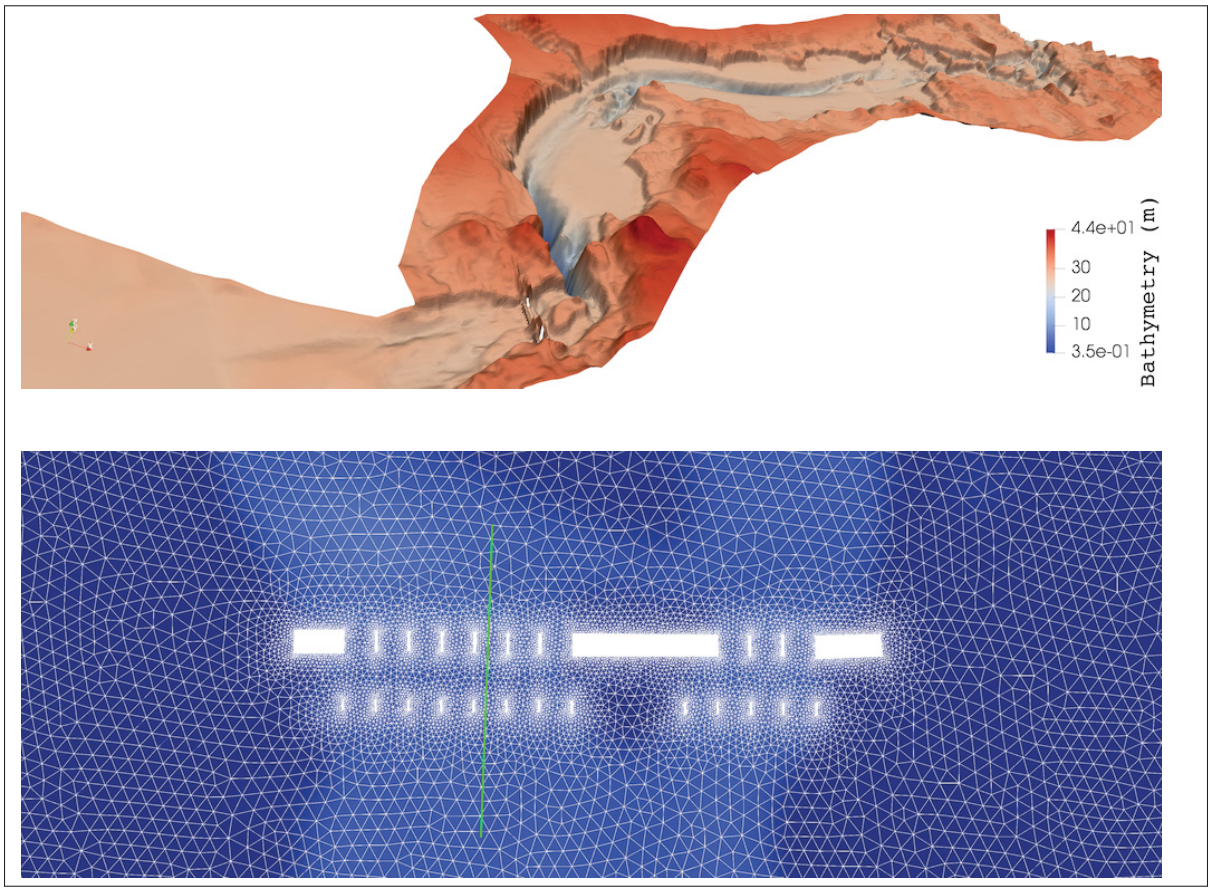
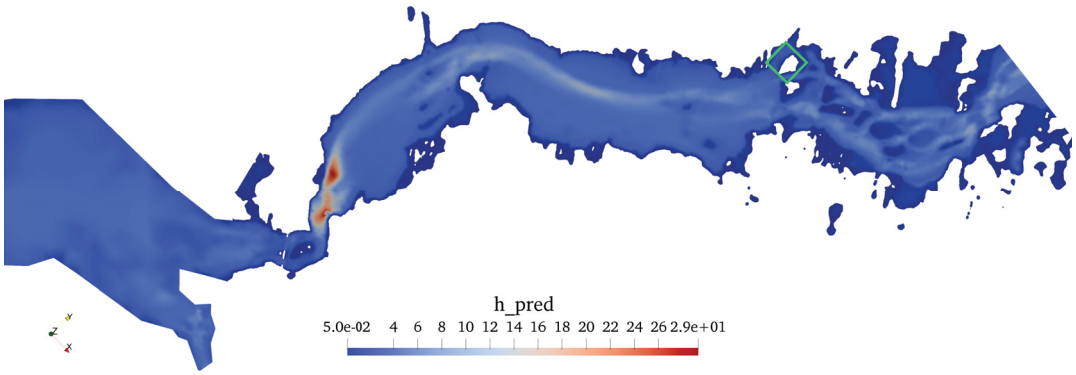


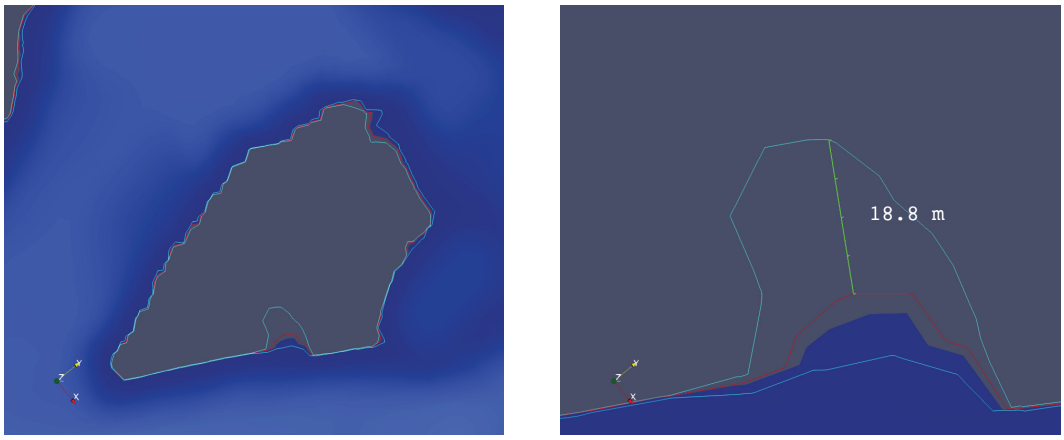
Figure 4.7 Setup for Milles-Iles river in Laval, QC, Canada. On top one can see the bathymetry, given by the *Communauté Métropolitaine de Montréal*, and at the bottom lies a portion of the triangle-based mesh, that features refinements abound the piers of a bridge

the usual point-estimate prediction of a regular network in the POD-NN framework is a step forward for a practical, engineering use.

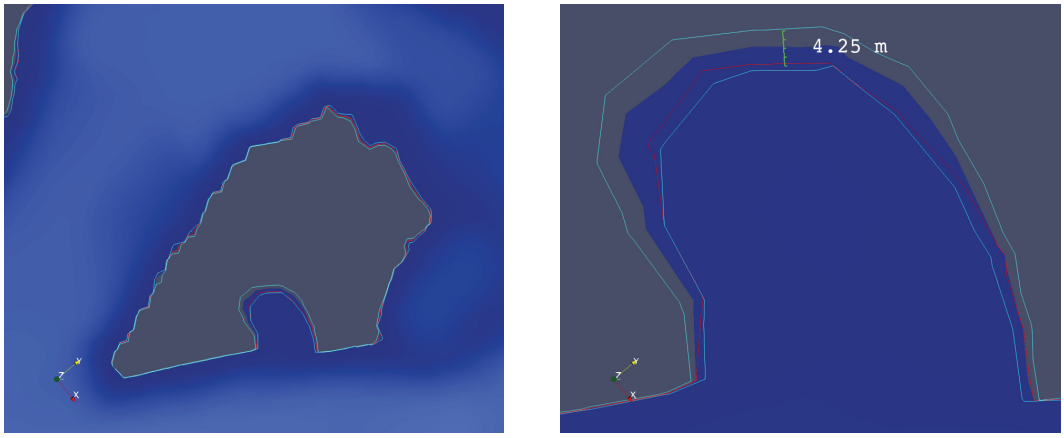
Finally, to make sure that our *out-of-distribution* predictions weren't just coincidences in the previous benchmarks, see Section 4.4.1 and 4.4.2, we also sampled new parameters from the whole $\Omega_{\text{out}} \cup \Omega$ domain, retrieved the mean across all DOFs of the predicted standard deviation, and rendered it in Figure 4.9. We observe that uncertainties snowball as soon as we exited the space where the model *knows*, validating the model, just as one could expect.



a) View from afar of a random test sample $Q = 884.4\ \mathrm{m}^3/\mathrm{s}$. This shows an iso-contour at $h = 0.05\ m$, with its boundary being the flooding lines. For illustration purposes, the overall predicted relative water height h_{pred} has been pictured throughout. The green boxes show the two different locations of the following closer shots



b) Random test sample with two levels of zoom, incoming flow of $Q = 884.4\ \mathrm{m}^3/\mathrm{s}$



c) Random test sample with two levels of zoom, incoming flow of $Q = 1159.8\ \mathrm{m}^3/\mathrm{s}$

Figure 4.8 POD-EnsNN application: flood modeling on the Mille-Iles river. Flooding lines at $h = 0.05\ m$ are shown on the close-up shots, with the red one for the CuteFlow solution, and the white ones representing the end of the predicted confidence interval $\pm 2\sigma_D$. The distance between the simulated value and the upper bound is measured

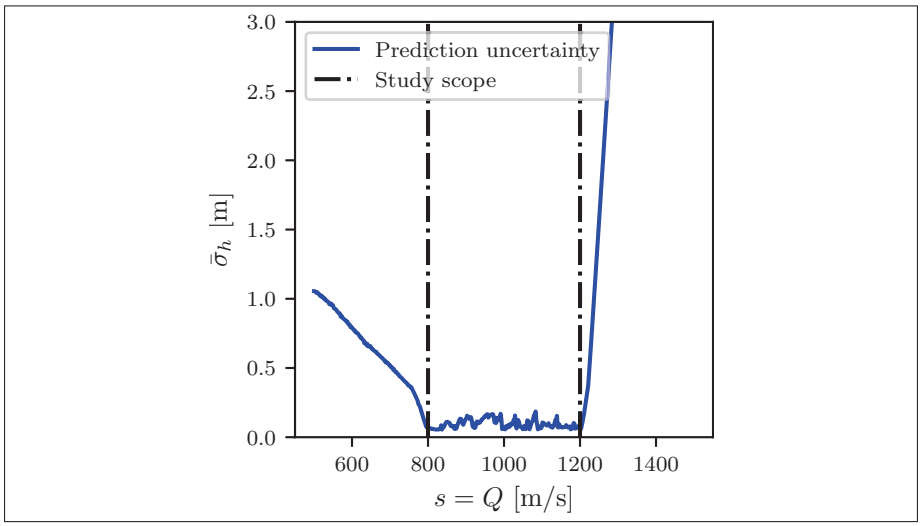


Figure 4.9 POD-EnsNN on the flooding case. Visualization of the average uncertainties for a range of inputs, with the two vertical black lines depicting the boundaries of the training and testing scope

4.5.5 Contribution to standard uncertainty propagation

Instead of considering the domain of the sampled inflow Ω as just a dataset, this is often used in the field as random inputs around a central, critical point for *uncertainty propagation* tasks, as performed in a similar context in Zokagoa & Soulaimani (2018). In this field, the use of a surrogate model is mandatory, since we wish to approximate the statistical moments of the output distributions to the model, i.e., the mean μ_{up} and the standard deviation σ_{up}.

On the flood modeling problem for the Milles-Iles river, the normal inflow is estimated to be of $Q_0 = 780$ m/s. Our snapshots have been sampled uniformly in $\Omega = [800, 1200]$ m/s, targeting a critical mean value of $Q_{\text{crit}} = 1000$ m/s, corresponding to an extreme flood discharge.

After having successfully trained and validated the model in Section 4.5.1, we now generate a new set of inputs $\boldsymbol{X}_{\text{up}}$ of size $N_{\text{up}} = 1,000,000$ uniformly on Ω. Running the full POD-EnsNN model, we obtain the outputs $\boldsymbol{U}_{\text{up}}$, with the quantity of interest being the water depth h here again. Since our model provide a local uncertainty for each sample point, we approximate the

statistical moments using the same mixture formulas as for sample prediction $(\mu_{*_i}, \sigma_{*_i})$,

$$\mu_{\text{up}} = \frac{1}{N_{\text{up}}} \sum_{i=1}^{N_{\text{up}}} \mu_{*_i}, \tag{4.32}$$

$$\sigma^2_{\text{up}} = \frac{1}{N_{\text{up}}} \sum_{i=1}^{N_{\text{up}}} (\sigma^2_{*_i} + \mu^2_{*_i}) - \mu^2_{\text{up}}. \tag{4.33}$$

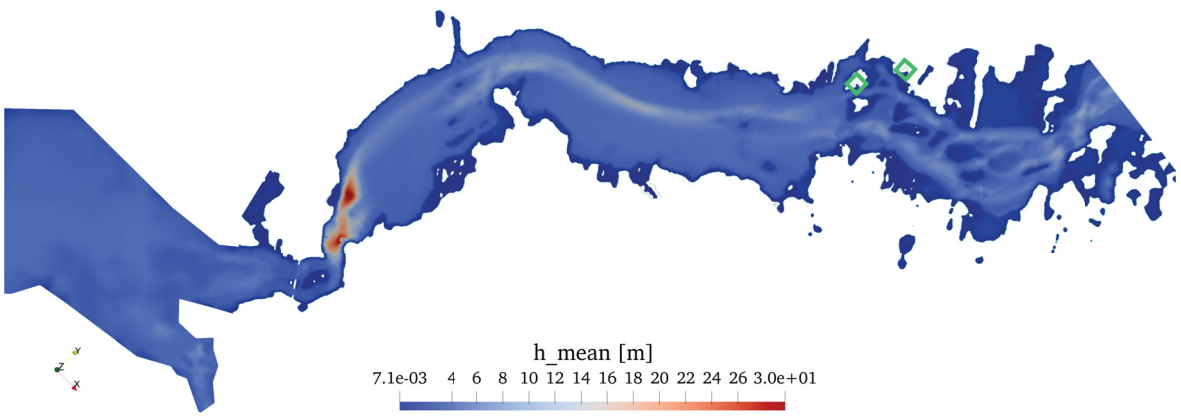
Additionally, we'll keep track of the regular statistical standard deviation σ_{ups} on the means, as a point of comparison, defined as

$$\sigma^2_{\text{ups}} = \frac{1}{N_{\text{up}}} \sum_{i=1}^{N_{\text{up}}} (\mu_{*_i} - \mu_{\text{up}})^2. \tag{4.34}$$

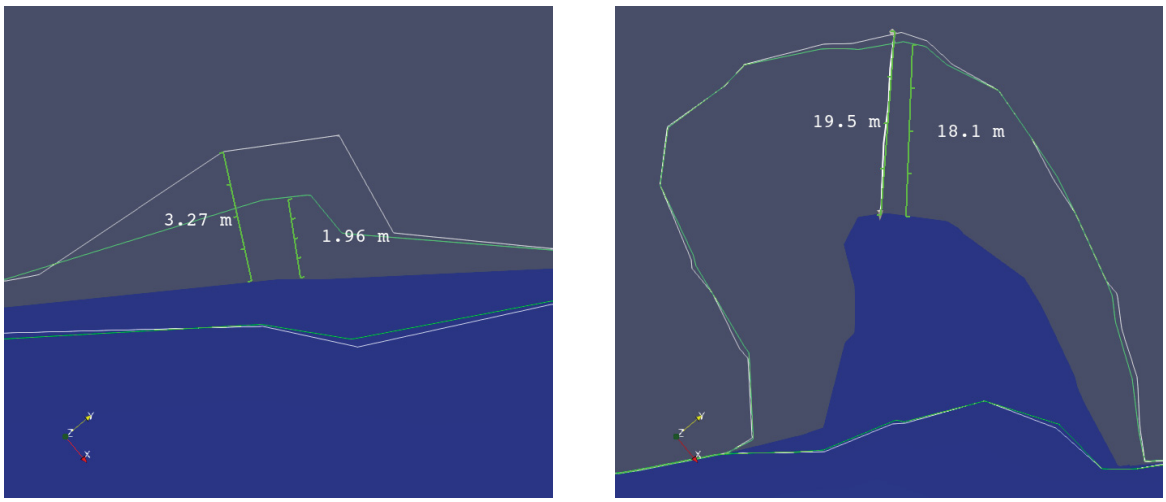
The produced probabilistic flooding map is depicted in Figure 4.10. On the very top, one can see a broad view of the flooding at $h = 0.05$ m, with the predicted $h_{\text{mean}} = \mu_{\text{up}}$ from (4.32), depicted as a colormap throughout, with two yellow boxes locating the two chosen close-up shots. These are displayed in the second row. On both, four lines on top of the mean blue water level: two green lines, showcasing two bands of the standard deviation over the predicted means only, $\pm 2\sigma_{\text{ups}}$, and two white lines, representing two bands of a standard deviation $\pm 2\sigma_{\text{up}}$ obtained averaging across each mean and variance predicted by the POD-EnsNN locally.
While these lines are very close throughout, as it is well represented by the second close-up shot on the right, where the difference measured is tiny, the gap sometimes increases, for instance, in the case of the first close-up shot, where the measured difference is much more significant. This attests to the potential usefulness of our approach is the realm of uncertainty propagation, effectively combining aleatoric (due to the distribution of Q) and epistemic (due to the numerical simulator). Nonetheless, epistemic uncertainty stays relatively minor in this case since averaging over the quite broad domain Ω mostly wipes away the predicted local variances.

4.5.6 An unsteady case: the failure of a dam

While the flooding predictions in the sense of generating flooded/non-flooded limits are a handy tool for public safety, it seemed interesting to apply the same framework to a time-dependent



a) View from afar of the mean over the whole predicted domain Ω. This shows an iso-contour at $h = 0.05\ m$, with its boundary being the flooding lines. For illustration purposes, the mean predicted relative water height h_{mean} has been pictured throughout. The yellow boxes show the two different locations of the following close-up shots



b) Two close-up shots, showing differences in the uncertainty around the mean water level (in blue)

Figure 4.10 POD-EnsNN for uncertainty propagation on the Milles-Iles river. Flooding lines at $h = 0.05\ m$ are shown on the close-up shots, with the green ones showing $\pm 2\sigma_{\text{ups}}$, the standard deviation over each predicted mean, while the white ones represent $\pm 2\sigma_{\text{up}}$, the approximation over each predicted mean and variance. Distances are measured between the mean, represented by the blue area, and each of these quantities

case: simulations result of a fictitious dam break on the same river, which is for interest also for dams owners.

The setup involves the same Shallow Water equations, as described in Section 4.5.1. The domain of study is a subdomain of the previous Ω_{xy}, with only $N_{xy} = 9734$ nodes and 18412 elements,

registering one degree of freedom per node, the water elevation η. Yet, for this case, we consider $N_t = 100$ time-steps, after the initial $t = 0$ s with $\Delta t = 0.1$ s between each step. $N_s = 20$ samples are considered for the non-spatial parameter: the water difference between the inflow and the outflow cross-sections at the moment of the dam break $s = \Delta\eta$, as displayed in Figure 4.4, sampled uniformly on $\Omega = [0, 3]\ m$.

As training hyperparameters, we settled on a number of epochs $N_e = 70,000$, a learning rate of 0.001, an L2 regularization of $\lambda = 0.01$, and adversarial training with a $\zeta = 0.001$ coefficient. POD was performed with $\epsilon = 1.10^{-8}$, producing $L = 52$ coefficients to be matched by half of the final layer, and the NN topology was 2 hidden layers of 140 neurons.

The training of each model took 2 minutes 23 seconds, 2 minutes 24 seconds, 2 minutes 25 seconds, 2 minutes 26 seconds, and 2 minutes 27 seconds on each GPU. The total, real time of the parallel process was 02 minutes and 49 seconds, reaching training losses $\mathcal{L} = 4.2265 \times 10^0$, 3.8916×10^0, 4.2662×10^0, 2.7526×10^0, and 2.3032×10^0, down from the initial $\mathcal{L}_0 = 1.2416 \times 10^5$, 1.0910×10^5, 1.1882×10^5, 1.2899×10^5, and 1.2625×10^5, also noted to picture the variability in the randomly initialized networks.

The overall relative errors reached were $RE_{\text{val}} = 0.06\%$ and $RE_{\text{tst}} = 0.11\%$, for validation and testing, respectively.

Results are displayed in Figure 4.11, where, from top to bottom, one can see representations of three time-steps, $t = 0\ s$, $t = 0.5\ s$, and $t = 2.0\ s$. On the left, a small colormap is displayed to comprehend the problem visually, with the investigated cross-section depicted as a white vector. We can see discontinuity of colors on the first one, that represents a dam, and in that case, a $\Delta\eta = 0.42\ m$ difference in the water levels, before the artificial collapse. The two subsequent time-steps picture the intense dynamics that follow. On the right, the cross-section is projected at each step, and we can see a decent approximation performed by the model, considering the high nonlinearity of the problem. It is, however, well depicted by the noticeable uncertainty associated, obtained from (4.18), and represented by the light blue area around the predicted blue line.

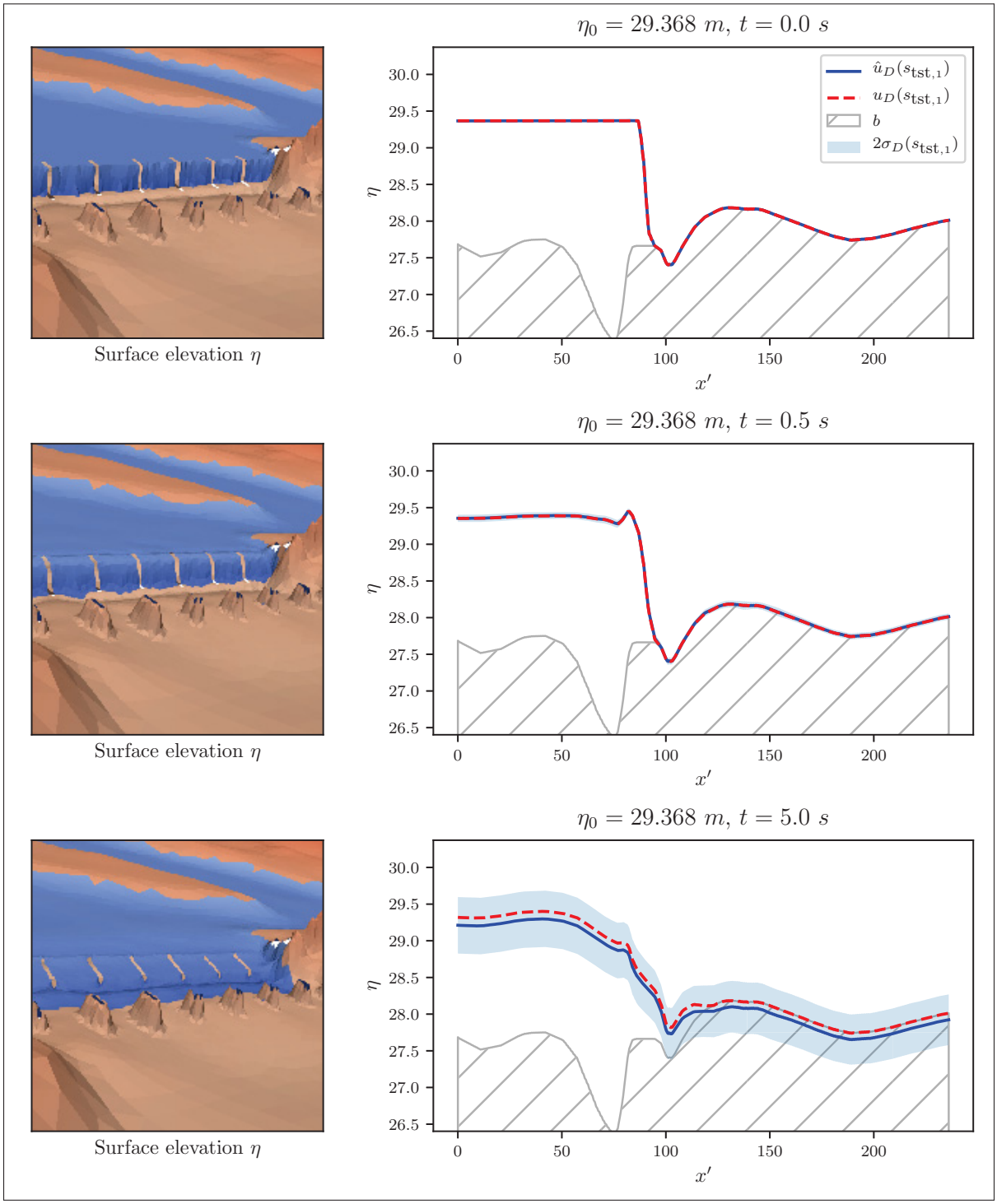


Figure 4.11 Left: color map according to η, showing the location of our cross-section x' (white vector). Right: plots of the water elevation on the cross-section of a random test snapshots on three time-steps, with the prediction $\hat{u}_D$, true value u_D, and confidence interval. The water in the river is flowing from right to left.

4.6 Exploring Bayesian Neural Networks

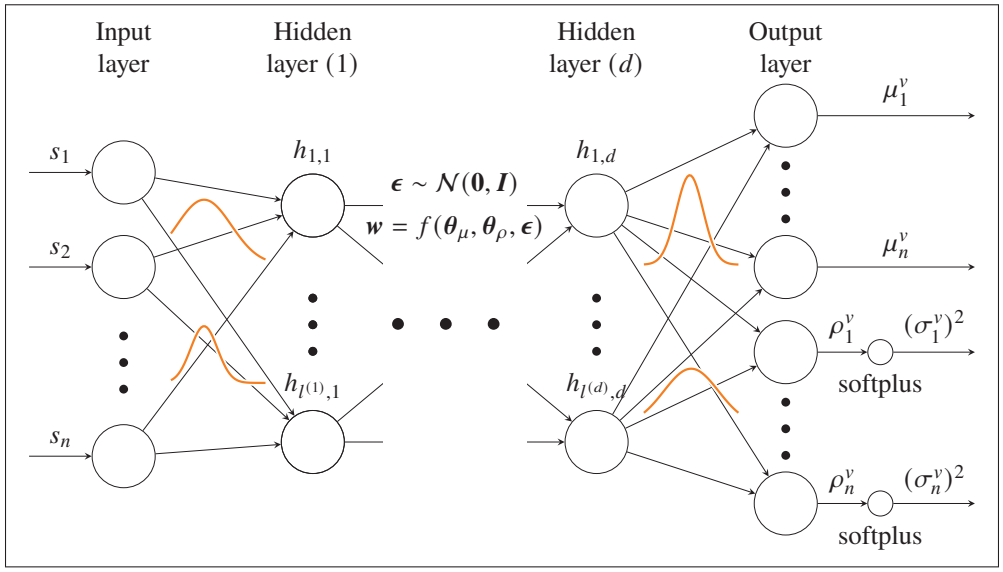


Figure 4.12 $\hat{u}_{DB}(\boldsymbol{X}; \boldsymbol{\theta}) \sim \mathcal{N}(\boldsymbol{\mu}^v, (\boldsymbol{\sigma}^v)^2)$, a probabilistic Bayesian Neural Network regression with a dual mean and variance output, and distributions on the weights

The other way to get to uncertainties in probabilities is to adopt the Bayesian view, yet lately, things have started getting better to include a fully Bayesian treatment within Deep Neural Networks, Blundell *et al.* (2015), designed to be compatible with backpropagation. In this section, we aim at implementing Bayesian Neural Networks within the POD-NN framework, which we will refer to as POD-BNN, and compare it to the Deep Ensembles approach.

4.6.1 Overview

To address the *aleatoric uncertainty*, Bayesian Neural Networks can make use of the same dual-output setting as the NNs we used earlier for Deep Ensembles, (μ, ρ), with the variance subsequently retrieved with the softplus function $\sigma = \log(1 + \exp(\rho))$.

But it's indeed in the *epistemic uncertainty* treatment that things are much different. Earlier, even though the NNs were providing us with a mean and variance, they were still deterministic, and variability was obtained by ensembling randomly initialized models.

On the contrary, the Bayesian treatment aims to assign distributions to the network's weights, and therefore have a probabilistic output by design, see Figure 4.12. In this context, one has to make multiple predictions, instead of numerous training times, to get data on uncertainties. Considering a dataset $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{y}\}$, the goal is to construct a likelihood function $p(\mathcal{D}|\boldsymbol{w})$, with $\boldsymbol{w}$ denoting both the weights $\boldsymbol{w}$ and the biases $\boldsymbol{b}$ for simplicity, so that we could achieve the following posterior predictive distribution

$$p(\boldsymbol{y}|\boldsymbol{X}, \mathcal{D}) = \int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}|\mathcal{D})\, d\boldsymbol{w}, \tag{4.35}$$

which is known to be intractable in a NN context, due to the infinite possibilities for the weights $\boldsymbol{w}$, leaving the posterior $p(\boldsymbol{w}|\mathcal{D})$ unknown as explained in Blundell *et al.* (2015).
There's, however, an approximate way to compute it, using Variational Inference, first presented by Hinton & van Camp (1993). The idea is to build up an approximation $q(\boldsymbol{w}|\boldsymbol{\theta})$ of $p(\boldsymbol{w}|\mathcal{D})$, and the goal is to minimize their Kullback-Leibler divergence, which measures the difference between them, denoted $\mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta}), ||p(\boldsymbol{w}|\mathcal{D}))$ w.r.t the new parameters $\boldsymbol{\theta}$, and expressed as

$$\begin{aligned}
\mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta})||p(\boldsymbol{w}|\mathcal{D})) &= \int q(\boldsymbol{w}|\boldsymbol{\theta}) \log \frac{q(\boldsymbol{w}|\boldsymbol{\theta})}{p(\boldsymbol{w}|\mathcal{D})}\, d\boldsymbol{w} && (4.36)\\
&= \mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta})||p(\boldsymbol{w})) - \mathbb{E}_{q(\boldsymbol{w}|\boldsymbol{\theta})} \log p(\mathcal{D}|\boldsymbol{w}) + \log p(\mathcal{D}) && (4.37)\\
&=: \mathcal{F}(\mathcal{D}, \boldsymbol{\theta}) + \log p(\mathcal{D}). && (4.38)
\end{aligned}$$

This term $\mathcal{F}(\mathcal{D}, \boldsymbol{\theta})$ we've just defined is commonly known as the *variational free energy*, and minimizing it w.r.t. to the weights doesn't involve the last term $\log p(\mathcal{D})$, so is equivalent to the goal of minimizing $\mathrm{KL}(q(\boldsymbol{w}|\boldsymbol{\theta}), ||p(\boldsymbol{w}|\mathcal{D}))$. Still according to Blundell *et al.* (2015), one can approximate it by drawing N_{mc} samples $\boldsymbol{w}^{(i)}$ from our new $q(\boldsymbol{w}|\boldsymbol{\theta})$, such as

$$\mathcal{F}(\mathcal{D}, \boldsymbol{\theta}) \approx \sum_{i=1}^{N} \left[\log q(\boldsymbol{w}^{(i)}|\boldsymbol{\theta}) - \log p(\boldsymbol{w}^{(i)}) - \log p(\mathcal{D}|\boldsymbol{w}^{(i)}) \right] := \mathcal{L}_{\mathrm{ELBO}}(\mathcal{D}, \boldsymbol{\theta}). \tag{4.39}$$

This defines our new loss function $\mathcal{L}_{\mathrm{ELBO}}$, named after the opposite of $\mathcal{F}(\mathcal{D}, \boldsymbol{\theta})$ usually known as the *Evidence Lower Bound* (ELBO).

4.6.2 Implementation

The idea behind the work of Blundell *et al.* (2015) was to have a fully Bayesian treatment of the weights while providing it in a compatible form to the usual *backpropagation* algorithm, mentioned in Section 4.3. One of the blockers is the forward pass that requires gradients to be tracked, in a way allowing their derivatives can then be backpropagated. In the j-th *variational layer*, we consider the weights and the biases to be parametrized by a distribution of mean $\boldsymbol{\theta}_{\mu}^{(j)}$ and raw variance $\boldsymbol{\theta}_{\rho}^{(j)}$. This setting leads the total number of trainable parameters of the network to be twice the one in a standard NN since each $\boldsymbol{w}^{(j)} = (\boldsymbol{\theta}_{\mu}^{(j)}, \boldsymbol{\theta}_{\rho}^{(j)})$.

In the forward pass, to keep track of the gradients, each operation has to be differentiable. We, therefore, construct a function $f(\boldsymbol{\theta}_{\mu}^{(j)}, \boldsymbol{\theta}_{\rho}^{(j)}, \boldsymbol{\epsilon}) = \boldsymbol{\theta}_{\mu}^{(j)} + \boldsymbol{\theta}_{\rho}^{(j)} \odot \boldsymbol{\epsilon}^{(j)}$, with $\boldsymbol{\epsilon}^{(j)}$ sampled from a parameter-free normal distribution, $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. It is known as the reparametrization trick, Kingma & Welling (2014).

The true variance on the weights $\boldsymbol{\theta}_{\sigma}^{(j)}$ isn't the direct parameter, but as earlier, we define it through a softplus function, with $\boldsymbol{\theta}_{\sigma}^{(j)} = \log(1 + \exp(\boldsymbol{\theta}_{\rho}^{(j)}))$.

The last step is to establish the Bayesian prior on the weights, $p(\boldsymbol{w})$. While there is plenty to choose from, whether they be *hardcoded* or *trainable*, for simplicity in this work we reuse the Gaussian mixture proposed in Blundell *et al.* (2015), defined for three positive hyperparameters π_0, π_1, and π_2, such as

$$p(\boldsymbol{w}) = \pi_0 \mathcal{N}(\boldsymbol{w}|0, \pi_1^2) + (1 - \pi_0)\mathcal{N}(\boldsymbol{w}|0, \pi_2^2). \tag{4.40}$$

The practical implementation steps for one training epoch are summarized in Algorithm 4.3. Predictions are made in a standard way, with the network being fed with the input data $\boldsymbol{X}$. However, calling it once will only trigger samples of the parameters $\boldsymbol{w}_b$; therefore, no two model calls will be identical. We hence require the use of an approximation in a simple Normal distribution over B different samples, using the same expressions as in Section 4.3 for the

averaged mean $\boldsymbol{\mu}_*^v$ and variance $\boldsymbol{\sigma}_*^{v2}$ in the reduced space

$$\boldsymbol{\mu}_*^v = \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{\mu}_{\boldsymbol{w}_b}^v, \tag{4.41}$$

$$\boldsymbol{\sigma}_*^{v2} = \frac{1}{B}\sum_{b=1}^{B} \left[(\boldsymbol{\sigma}_{\boldsymbol{w}_b}^v)^2 + (\boldsymbol{\mu}_{\boldsymbol{w}_b}^v)^2\right] - \boldsymbol{\mu}_*^2. \tag{4.42}$$

Algorithm 4.3 Epoch training of a BNN

Feed the model with the dataset $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$
for *each variational layer* $1 \leq j \leq d$ **do**
  $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
  $\boldsymbol{w}^{(j)} = f(\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\rho^{(j)}, \boldsymbol{\epsilon}^{(j)})$
  $\boldsymbol{\theta}_\sigma^{(j)} = \text{softplus}(\boldsymbol{\theta}_\rho^{(j)})$
  Sample the variational posterior $q(\boldsymbol{w}^{(j)}|\boldsymbol{\theta}^{(j)}) = \mathcal{N}(\boldsymbol{w}^{(j)}|\boldsymbol{\theta}_\mu^{(j)}, \boldsymbol{\theta}_\sigma^{(j)})$
  Sample the prior $p(\boldsymbol{w}^{(j)})$
  Contribute the posterior and prior to the loss,
   $\mathcal{L}_{\text{ELBO}}$ += $\log q(\boldsymbol{w}^{(j)}|\boldsymbol{\theta}^{(j)}) + \log p(\boldsymbol{w}^{(j)})$
  Perform the forward pass $\boldsymbol{h}^{(j)} = \phi(\boldsymbol{w}^{(j)}\boldsymbol{h}^{(j-1)} + \boldsymbol{b}^{(j)})$
end
Retrieve the outputs $\boldsymbol{\mu}^v, \boldsymbol{\sigma}^{v2}$ from the NN
Compute the likelihood from the ouputs, $p(\mathcal{D}|\boldsymbol{w}) \sim \mathcal{N}(\boldsymbol{\mu}^v, \boldsymbol{\sigma}^{v2})$
Contribute the NLL to the loss, $\mathcal{L}_{\text{ELBO}}$ += $-\log p(\mathcal{D}|\boldsymbol{w})$
Backpropagate the gradients $\dfrac{\partial \mathcal{L}_{\text{ELBO}}}{\partial \boldsymbol{\theta}}$ to update the weights

4.6.3 Setup and results

A Bayesian Neural Network approach is now applied to perform the regression and create the surrogate model $\hat{u}_{DB}$ in the POD-NN framework, and the two test cases are the Ackley Function (2D), see Section 4.4.1, and the Flood Predictions on the Mille-Iles river, Section 4.5.1.

The same datasets $\mathcal{D} = \{\boldsymbol{X}, \boldsymbol{v}\}$ are used on each case, and the setup remains the same. The prior is chosen to have the parameters $\pi_0 = 0.5$ and $\pi_2 = 0.1$ for both cases, while we picked $\pi_1 = 4.0$ for Ackley and $\pi_1 = 2.0$ for the flood modeling predictions.

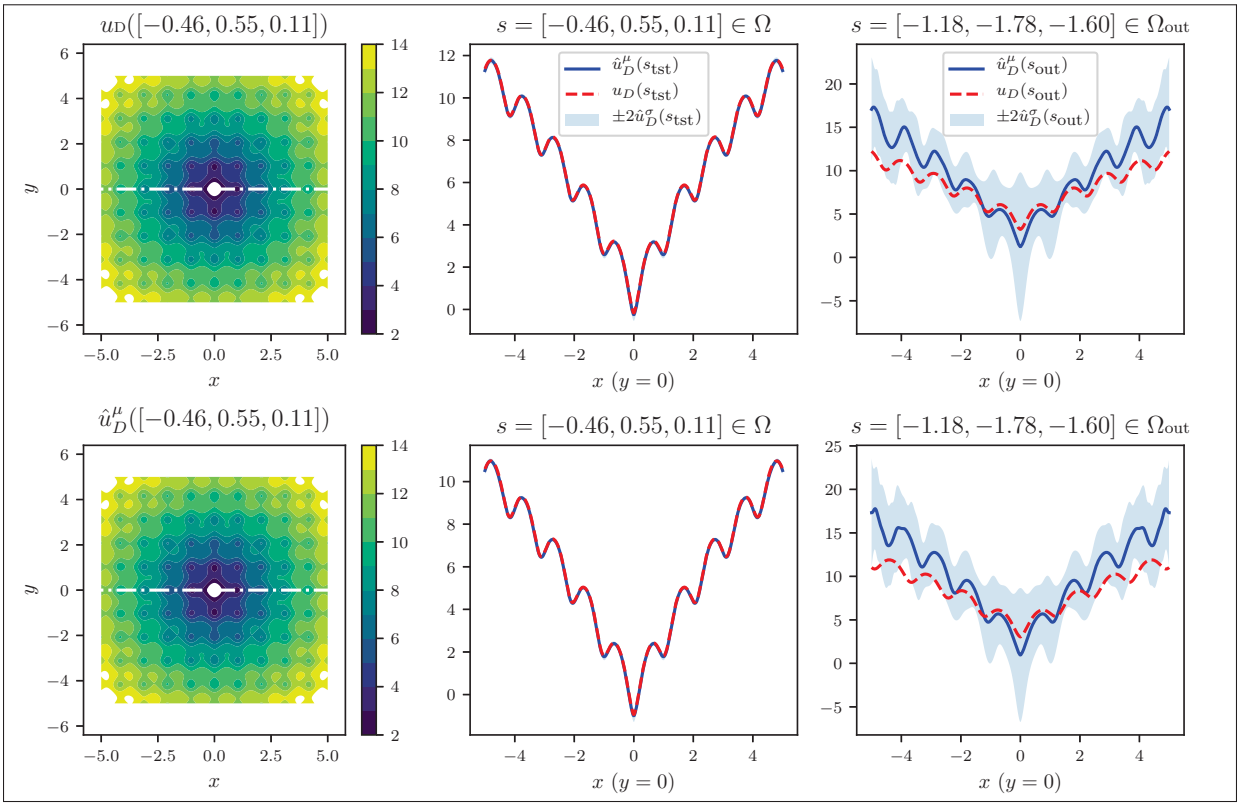


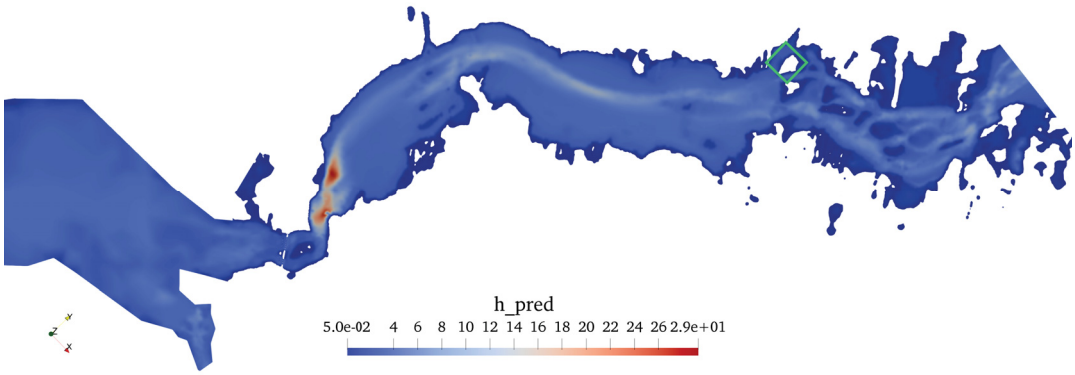
Figure 4.13 Identical setup as Figure 4.2, second column samples in the scope and third column out-of-distribution, yet with Bayesian Neural Network regression

The trainable parameters $\boldsymbol{\theta}^{(j)}$ (weight or bias) of the j-th layer have been randomly initialized, with

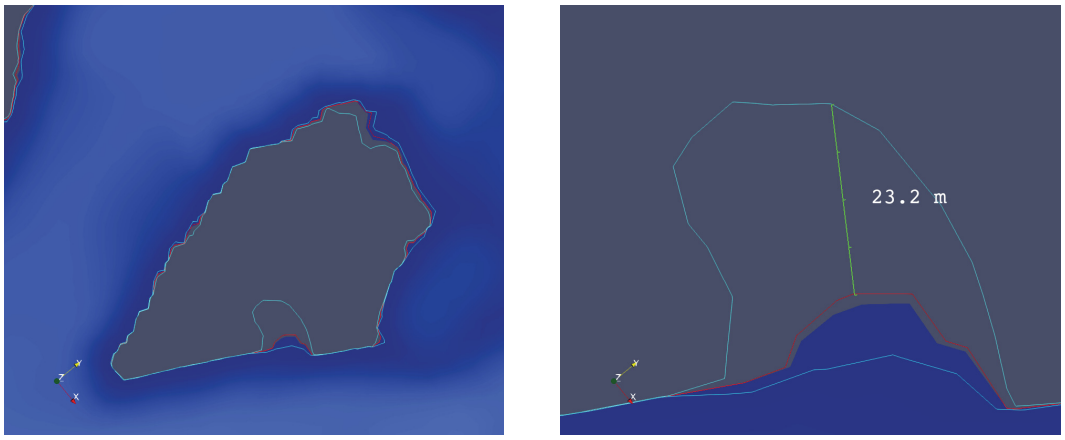
$$\boldsymbol{\theta}^{(j)} = (\boldsymbol{\theta}_{\mu}^{(j)}, \boldsymbol{\theta}_{\sigma}^{(j)}) \sim \mathcal{N}\left(\mathbf{0}, \sqrt{\pi_0 \pi_1^2 + (1-\pi)\pi_2^2}\boldsymbol{I}\right). \tag{4.43}$$

The results of performing the regression using Bayesian Neural Networks are showcased in Figure 4.13 and 4.14, and are very similar to the one conducted in Section 4.4, yet only requires one training time.

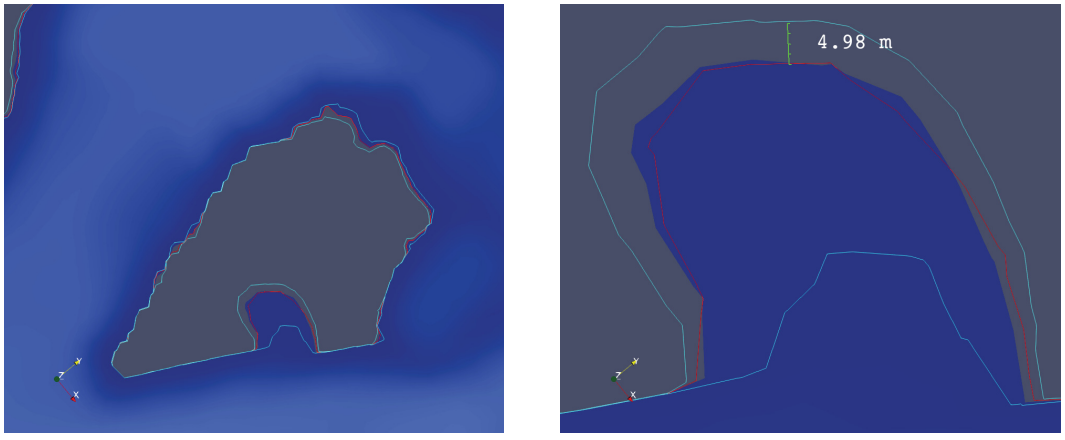
The Ackley function case took 5 minutes and 24 seconds on a standard desktop CPU to reach a relative validation error of $RE_{\text{val}} = 0.04\%$ after $N_e = 70,000$ training epochs, and $RE_{\text{tst}} = 0.03\%$ for the test error. One can see that the slices shown in Figure 4.13 are very similar to the ones performed by the POD-EnsNN approach in Figure 4.2, with a similar proper fitting of the predicted mean, and a comparable uncertainty handling, hence validating the framework.



a) View from afar of a random test sample $Q = 884.4\ \mathrm{m}^3/\mathrm{s}$. This shows an iso-contour at $h = 0.05\ m$, with its boundary being the flooding lines. For illustration purposes, the overall predicted relative water height h_{pred} has been pictured throughout. The green boxes show the two different locations of the following closer shots



b) Random test sample with two levels of zoom, incoming flow of $Q = 884.4\ \mathrm{m}^3/\mathrm{s}$



c) Random test sample with two levels of zoom, incoming flow of $Q = 1159.8\ \mathrm{m}^3/\mathrm{s}$

Figure 4.14 POD-BNN application: flood modeling on the Mille-Iles river. Flooding lines at $h = 0.05\ m$ are shown on the close-up shots, with the red one for the CuteFlow solution, and the white ones representing the end of the predicted confidence interval $\pm 2\sigma_D$. The distance between the simulated value and the upper bound is measured

For the flood modeling case, the application of foremost interest, the training took 11 minutes and 36 seconds on a standard desktop CPU to reach a relative validation error of $RE_{\text{val}} = 0.25\%$ after $N_e = 200,000$ training epochs, and $RE_{\text{tst}} = 0.26\%$ as well for the test error. These computing times are similar to the ensembles approach in a linear training setting. However, when parallelized as we did in Section 4.5.1, ensembles were faster by a factor of at least 5.

Figure 4.14 shows similar probabilistic flooding lines as Figure 4.8, yet the Bayesian approach seems to produce larger confidence intervals, as it is depicted quantitatively by the measured distance on the second column between the CuteFlow solution and the upper bound of the confidence interval.

As for the POD-EnsNN, we inspected the uncertainties predicted out-of-distribution, and the results are displayed in Figure 4.15. Nonetheless, one can note that the increase isn't as drastic as for the Deep Ensembles, and our observations were that the prior parameters had a direct impact on this matter.

Documented source code will be made available at https://github.com/pierremtb/POD-UQNN, on the `POD-BNN` branch.

Additionally, quality training was harder to achieve for Bayesian Neural Networks compared to Deep Ensembles, since the convergence itself greatly depended in our experiments on numerical stability issues such as the coefficients to be inserted before or in the softplus function when ensuring positivity. The probabilistic layers and their uncertainty prediction was also dependent on the number of training epochs, with cases of uncertainties dropping when more epochs were performed. More practical details and handy insights in the implementation of BNN are discussed in Keydana (2019), with examples in the R programming language.

4.7 Wrapping up

The excellent regression power of Deep Neural Networks is an asset to play along with Proper Orthogonal Decomposition to build reduced-order models. Through 1D and 2D benchmarks, we've shown that the simplicity of the approach didn't prevent us from getting great results in terms of accuracy, and the training times were very decent, even on regular computers—time-dependent problems, however, require the use of GPUs to speed up training.

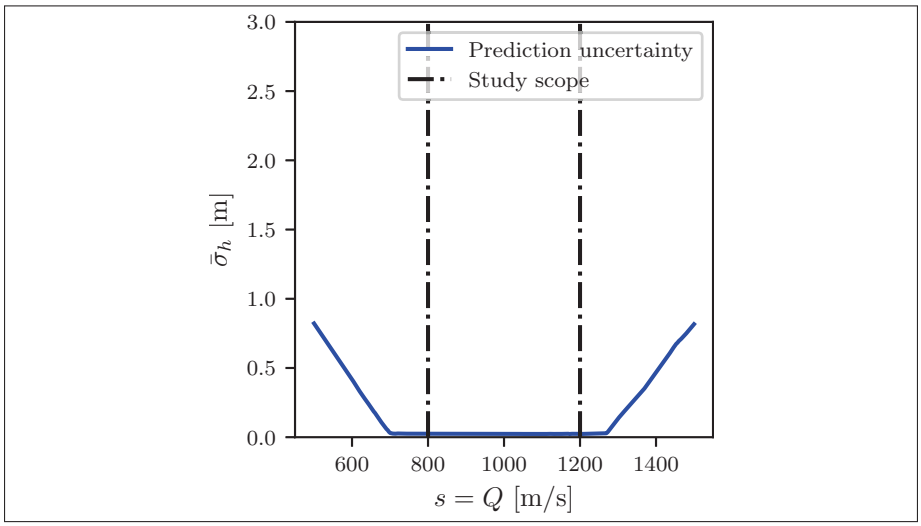


Figure 4.15 POD-BNN on the flooding case. Visualization of the average uncertainties for a range of inputs, with the two vertical black lines depicting the boundaries of the training and testing scope

It has also been shown that while the standard NNs were rapidly predicting inaccurate quantities when brought out of the training scope, adopting an uncertainty-enabled approach kept the true values within the confidence interval, and having it growing larger makes up for a great warning. Bayesian Neural Networks, as introduced by Blundell *et al.* (2015), have also been explored, as a way to bundle all the uncertainty quantification within the model, hence avoiding ensembling. Future work will focus on stabilizing the Bayesian Neural Networks approach, which still requires a much finer tuning compared to the flexibility of Deep Ensembles, and applying it both to refined meshes, that will require the POD step to be performed on a subdomain basis to avoid memory issues, to better assess the performance of our uncertainties-aware POD-NN framework in a more complicated, real-life engineering problem. The work being done in Latent-Variable Models, such as Zhu & Zabaras (2018), are also a different way to think about reduced-order modeling and deserves more in-depth exploration.

Flood modeling itself in the context of the Milles-Iles river and beyond provides many future exploration directions since various other parameters have a direct influence on the results, such as the Manning roughness of the bed, as well as its elevation, also complicated by measurement uncertainties.

CONCLUSION AND RECOMMENDATIONS

The last decade has seen tremendous growth in the Machine Learning and Artificial Intelligence power, and it's becoming a game-changer in more and more fields. The tools are becoming rock-solid, as is TensorFlow, Abadi (2016), to mention only one, and while they have the potential to revolutionize more and more industries, they are nowadays a must-have for anyone interested in computational science.

While we started off investigating the brand new field of Physics-Informed Machine Learning which had many breakthroughs in the use of specific network training tools like automatic differentiation for older problems and therefore removing the need for numerical differentiation, Raissi *et al.* (2019a), we had to pivot away to stay purely non-intrusive, because the very nature of this branch has to do with encoding the laws of physics in the model definition, which mismatched with our purpose.

In this work, we built upon the work of other pioneers in the mixing of Machine Leaning techniques with more conventional Fluid Mechanics simulation methods. The POD-NN framework first presented in Hesthaven & Ubbiali (2018) and extended in Wang *et al.* (2019) is leveraging the tremendous nonlinear, complex regression power of Deep Neural Networks to make sense of and complete the results of the well-known order-reducing Proper Orthogonal Decomposition method.

After properly defining its implementation and testing it on both steady and time-dependent benchmark problems, we tried to implement novel probabilistic ideas to make it aware of uncertainties and fulfill our objective of having a model that *knows when it doesn't*. While we first settled on Deep Ensembles from Lakshminarayanan *et al.* (2017) for our first attempt in building an uncertainties-aware version of the POD-NN framework, further exploration has been performed in the realm of Bayesian Neural Networks, in the frame of Blundell *et al.* (2015). They, however, proved to be very tricky to train and incredibly sensible to hyperparameters and would require more in-depth work to be fully implemented in a reliable way for our purpose.

The Milles-Iles river located in Laval, QC, Canada, has been our playground for multiple use cases of the two different approaches presented in this work, the POD-EnsNN model implementing Deep Ensembles, and POD-BNN model featuring Bayesian Neural Networks. We started from a simple one-dimensional shock-wave test case problem to both attest to the solver we've later used a source of data, CuteFlow, and our statistical approach. The problem has been extended to a dam break scenario on the real river topology, as well as probabilistic flooding map generation. This is where the approach shines since our models are capable of producing flooding lines within a predicted confidence interval, either in a local prediction manner, such as a real-time context where these lines need to be computed for a new parameter, or in a more global, uncertainty propagation case, where we think of an unknown extreme and critical inflow, for which one wishes to assess the consequences of profound changes in this quantity. And instead of computing the statistical moments of the output distribution from the point estimates of a surrogate model such as a standard Neural Network, our model is considering the contribution of each local uncertainty and, therefore, producing a more extensive and safer confidence area around the predicted flooding line.

Additional environmental parameters should be considered and investigated in the approach to build even more realistic models, such as the bed elevation or roughness, from which only uncertainty-prone measurements are available. And for such large scale scenarii, the curse of dimensionality, as mentioned in Chapter 1, may eventually play a role and will have to be taken into consideration.

Other Machine Learning methods such as Latent-Variable Models, reviewed in Li & Chen (2016), or Bayesian Encoder-Decoders from Zhu & Zabaras (2018) are two future directions to work with, that also would, in a sense, be conceptually similar to bundling our compression-decompression flow inside the model. Very recently, work has also been performed on implementing the Bayesian view in Physics Informed Neural Networks, Yang, Meng & Karniadakis (2020). Bridging the gap between POD-based approaches and the novel physics-informed methods, or

physics-constrained, as lately introduced in Magiera, Ray, Hesthaven & Rohde (2020), would also represent a big step forward.

REFERENCES

Abadi, M. (2016). TensorFlow: A System for Large-Scale Machine Learning. Consulted at https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Abdedou, A. & Soulaimani, A. (2018). A non-intrusive B-splines Bézier elements-based method for uncertainty propagation. *Computer Methods in Applied Mechanics and Engineering*, 345. doi: 10.1016/j.cma.2018.10.047.

Abraham, S., Tsirikoglou, P., Miranda, J., Lacor, C., Contino, F. & Ghorbaniasl, G. (2018). Spectral representation of stochastic field data using sparse polynomial chaos expansions. *Journal of Computational Physics*, 367, 109–120. doi: 10.1016/J.JCP.2018.04.025.

Ahrens, J., Geveci, B. & Law, C. (2005). Paraview: An end-user tool for large data visualization.

Amsallem, D. & Farhat, C. (2014). On the Stability of Reduced-Order Linearized Computational Fluid Dynamics Models Based on POD and Galerkin Projection: Descriptor vs Non-Descriptor Forms. In *Reduced Order Methods for Modeling and Computational Reduction* (pp. 215–233). Springer International Publishing. doi: 10.1007/978-3-319-02090-7_8.

Bailey, K. (2016). Gaussian Processes for Dummies.

Barber, D. & Bishop, C. (1998). Ensemble learning in Bayesian neural networks. *Nato ASI Series F Computer and Systems Sciences*, (Bishop 1995), 215–237.

Barthelmann, V., Novak, E. & Ritter, K. (2000). High dimensional polynomial interpolation on sparse grids. *Advances in Computational Mathematics*, 12(4), 273–288.

Basdevant, C., Deville, M., Haldenwang, P., Lacroix, J. M., Ouazzani, J., Peyret, R., Orlandi, P. & Patera, A. T. (1986). Spectral and finite difference solutions of the Burgers equation. *Computers & fluids*, 14(1), 23–41.

Bellman, R. (1966). Dynamic programming. *Science*, 153(3731), 34–37.

Benner, P., Gugercin, S. & Willcox, K. (2015). A survey of projection-based model reduction methods for parametric dynamical systems. *SIAM Review*, 57(4), 483–531. doi: 10.1137/130932715.

Bishop, C. M. (2006). *Pattern Recogniton and Machine Learning*. Springer-Verlag New York. Consulted at https://www.springer.com/gp/book/9780387310732http://users.isr.ist.utl.pt/{~}wurmd/Livros/school/Bishop-PatternRecognitionAndMachineLearning-Springer2006.pdf.

Blundell, C., Cornebise, J., Kavukcuoglu, K. & Wierstra, D. (2015). Weight Uncertainty in Neural Networks. *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, (ICML'15), 1613–1622.

Brunton, S. L. & Kutz, J. N. (2019). *Data-Driven Science and Engineering*. Cambridge University Press. doi: 10.1017/9781108380690.

Brunton, S. L., Proctor, J. L. & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15), 3932–3937. doi: 10.1073/PNAS.1517384113.

Burkardt, J., Gunzburger, M. & Lee, H. C. (2006). Centroidal voronoi tessellation-based reduced-order modeling of complex systems. *SIAM Journal on Scientific Computing*, 28(2), 459–484. doi: 10.1137/5106482750342221x.

Carlberg, K. T., Jameson, A., Kochenderfer, M. J., Morton, J., Peng, L. & Witherden, F. D. (2019). Recovering missing CFD data for high-order discretizations using deep neural networks and dynamics learning. *Journal of Computational Physics*, 395, 105–124. doi: 10.1016/J.JCP.2019.05.041.

Couplet, M., Basdevant, C. & Sagaut, P. (2005). Calibrated reduced-order POD-Galerkin system for fluid flow modelling. *Journal of Computational Physics*, 207(1), 192–220. doi: 10.1016/J.JCP.2005.01.008.

Després, B. & Jourdren, H. (2020). Machine Learning design of Volume of Fluid schemes for compressible flows. *Journal of Computational Physics*, 408, 109275. doi: 10.1016/J.JCP.2020.109275.

Galland, J.-C., Goutal, N. & Hervouet, J.-M. (1991). TELEMAC: A new numerical model for solving shallow water equations. *Advances in Water Resources*, 14(3), 138–148. doi: 10.1016/0309-1708(91)90006-A.

Ghanem, R. G. & Spanos, P. D. (1991). Stochastic finite element method: Response statistics. In *Stochastic finite elements: a spectral approach* (pp. 101–119). Springer.

Glorot, X. & Bengio, Y. (2010). *Understanding the difficulty of training deep feedforward neural networks*. Consulted at http://www.iro.umontreal.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. MIT Press.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. & Bengio, Y. (2014a). Generative Adversarial Networks. Consulted at http://arxiv.org/abs/1406.2661.

Goodfellow, I. J., Shlens, J. & Szegedy, C. (2014b). Explaining and Harnessing Adversarial Examples. Consulted at http://arxiv.org/abs/1412.6572.

Graves, A. (2011). Practical Variational Inference for Neural Networks. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F. & Weinberger, K. Q. (Eds.), *Advances in Neural*

Information Processing Systems 24 (pp. 2348–2356). Curran Associates, Inc. Consulted at http://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks.pdf.

Guo, M. & Hesthaven, J. S. (2018). Reduced order modeling for nonlinear structural analysis using Gaussian process regression. *Computer Methods in Applied Mechanics and Engineering*, 341, 807–826. doi: 10.1016/J.CMA.2018.07.017.

Hanna, B. N., Dinh, N. T., Youngblood, R. W. & Bolotnov, I. A. (2020). Machine-learning based error prediction approach for coarse-grid Computational Fluid Dynamics (CG-CFD). *Progress in Nuclear Energy*, 118, 103140. doi: 10.1016/J.PNUCENE.2019.103140.

Hernandez-Lobato, J. M. & Adams, R. (2015). Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks. *Proceedings of the 32nd International Conference on Machine Learning*, 37(Proceedings of Machine Learning Research), 1861–1869. Consulted at http://proceedings.mlr.press/v37/hernandez-lobatoc15.html.

Hesthaven, J. & Ubbiali, S. (2018). Non-intrusive reduced order modeling of nonlinear problems using neural networks. *Journal of Computational Physics*, 363, 55–78. doi: 10.1016/J.JCP.2018.02.037.

Hinton, G. E. (2007). Learning multiple layers of representation. *Trends in Cognitive Sciences*, 11(10), 428–434. doi: 10.1016/J.TICS.2007.09.004.

Hinton, G. E. & van Camp, D. (1993). Keeping the Neural Networks Simple by Minimizing the Description Length of the Weights. *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, (COLT '93), 5–13. doi: 10.1145/168304.168306.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780. doi: 10.1162/neco.1997.9.8.1735.

Holmes, P., Lumley, J. L. & Berkooz, G. (1996). *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*. Cambridge University Press. doi: 10.1017/CBO9780511622700.

Holmes, P. J., Lumley, J. L., Berkooz, G., Mattingly, J. C. & Wittenberg, R. W. (1997). Low-dimensional models of coherent structures in turbulence. *Physics Report*, 287(4), 337–384. doi: 10.1016/S0370-1573(97)00017-3.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8), 2554–2558. doi: 10.1073/pnas.79.8.2554.

Hsieh, W. W. (2009). *Machine Learning Methods in the Environmental Sciences: Neural Networks and Kernels*. Cambridge University Press. Consulted at https://www.xarg.org/ref/a/0521791928/.

Hu, R., Fang, F., Pain, C. & Navon, I. (2019). Rapid spatio-temporal flood prediction and uncertainty quantification using a deep learning method. *Journal of Hydrology*, 575, 911–920. doi: 10.1016/J.JHYDROL.2019.05.087.

Jollife, I. T. & Cadima, J. (2016). *Principal component analysis: A review and recent developments*. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences. Royal Society of London. doi: 10.1098/rsta.2015.0202.

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J. & Aila, T. (2019). Analyzing and improving the image quality of stylegan. *arXiv preprint arXiv:1912.04958*.

Kendall, A. & Gal, Y. (2017). What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision? *Advances in Neural Information Processing Systems 30*, pp. 5574–5584. Consulted at http://papers.nips.cc/paper/7141-what-uncertainties-do-we-need-in-bayesian-deep-learning-for-computer-vision.pdf.

Keydana, S. (2019). TensorFlow for R: Adding uncertainty estimates to Keras models with tfprobability. Consulted at https://blogs.rstudio.com/tensorflow/posts/2019-06-05-uncertainty-estimates-tfprobability/.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P. & Welling, M. (2014, dec). Auto-encoding variational bayes. *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*.

Krasser, M. (2019). Variational inference in Bayesian neural networks - Martin Krasser's Blog. Consulted at http://krasserm.github.io/2019/03/14/bayesian-neural-networks/.

Krogh, A. & Hertz, J. A. (1992). A Simple Weight Decay Can Improve Generalization. In Moody, J. E., Hanson, S. J. & Lippmann, R. P. (Eds.), *Advances in Neural Information Processing Systems 4* (pp. 950–957). Morgan-Kaufmann. Consulted at http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf.

Kutz, J. N. (2017). Deep learning in fluid dynamics. *Journal of Fluid Mechanics*, 814, 1–4. doi: 10.1017/jfm.2016.803.

Lakshminarayanan, B., Pritzel, A. & Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in Neural Information Processing Systems*, pp. 6402–6413.

Lam, S. K., Pitrou, A. & Seibert, S. (2015). Numba: A LLVM-based Python JIT Compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, (LLVM '15), 7:1—-7:6. doi: 10.1145/2833157.2833162.

Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J. & Sohl-Dickstein, J. (2017). Deep Neural Networks as Gaussian Processes. Consulted at http://arxiv.org/abs/1711.00165.

Li, P. & Chen, S. (2016). A review on Gaussian Process Latent Variable Models. *CAAI Transactions on Intelligence Technology*, 1(4), 366–376. doi: 10.1016/J.TRIT.2016.11.004.

Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2), 146–160. doi: 10.1007/BF01931367.

Liu, D. C. & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1), 503–528. doi: 10.1007/BF01589116.

Lu, Z., Pu, H., Wang, F., Hu, Z. & Wang, L. (2017). The Expressive Power of Neural Networks: A View from the Width. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S. & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems 30* (pp. 6231–6239). Curran Associates, Inc. Consulted at http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

Lumley, J. L. The structure of inhomogeneous turbulence. in atmospheric turbulence and wave propagation. Ed. AM Yaglom, VI Tatarski, 1967. Moscow: Nauka.

Mackay, D. J. C. (1995). Probable networks and plausible predictions — a review of practical Bayesian methods for supervised neural networks. *Network: Computation in Neural Systems*, 6(3), 469–505. doi: 10.1088/0954-898X_6_3_011.

Magiera, J., Ray, D., Hesthaven, J. S. & Rohde, C. (2020). Constraint-aware neural networks for Riemann problems. *Journal of Computational Physics*, 409, 109345. doi: 10.1016/J.JCP.2020.109345.

Maleewong, M. & Sirisup, S. (2011). On-line and off-line POD assisted projective integral for non-linear problems: A case study with burgers' equation. *World Academy of Science, Engineering and Technology*, 79(7), 952–960.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. & Dean, J. (2013). Distributed Representations of Words and Phrases and Their Compositionality. *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, (NIPS'13), 3111–3119.

Nair, V. & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on International Conference on Machine Learning*, (ICML'10), 807–814.

Nix, D. A. & Weigend, A. S. (1994). Estimating the mean and variance of the target probability distribution. *Proceedings of 1994 ieee international conference on neural networks (ICNN'94)*, 1, 55–60.

Pang, G., Yang, L. & Karniadakis, G. E. (2019). Neural-net-induced Gaussian process regression for function approximation and PDE solution. *Journal of Computational Physics*, 384, 270–288. doi: 10.1016/J.JCP.2019.01.045.

Pearson, K. (1901). LIII. On lines and planes of closest fit to systems of points in space . *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11), 559–572. doi: 10.1080/14786440109462720.

Raissi, M., Perdikaris, P. & Karniadakis, G. (2019a). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving non-linear partial differential equations. *Journal of Computational Physics*, 378, 686–707. doi: 10.1016/J.JCP.2018.10.045.

Raissi, M., Perdikaris, P. & Karniadakis, G. E. (2017a). Machine learning of linear differential equations using Gaussian processes. *Journal of Computational Physics*, 348, 683–693. doi: 10.1016/J.JCP.2017.07.050.

Raissi, M., Perdikaris, P. & Karniadakis, G. E. (2017b). Numerical Gaussian Processes for Time-dependent and Non-linear Partial Differential Equations. *SIAM Journal on Scientific Computing*, 40(1), A172–A198. doi: 10.1137/17M1120762.

Raissi, M., Wang, Z., Triantafyllou, M. S. & Karniadakis, G. E. (2019b). Deep learning of vortex-induced vibrations. *Journal of Fluid Mechanics*, 861, 119–137. doi: 10.1017/jfm.2018.872.

Rasmussen, C. E. & Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press. Consulted at http://www.gaussianprocess.org/gpml/.

Rina Dechter. (1986). Learning While Searching In Constraint-Satisfaction-Problems. *Annals of Mathematics*, 178 – 183. Consulted at www.aaai.org.

Rosenblatt, F. (1958). *THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN 1* (Report n°6).

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536. doi: 10.1038/323533a0.

Sarkar, A. & Ghanem, R. (2002). Mid-frequency structural dynamics with parameter uncertainty. *Computer Methods in Applied Mechanics and Engineering*, 191(47), 5499–5513. doi: https://doi.org/10.1016/S0045-7825(02)00465-6.

Sergeev, A. & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. Consulted at https://arxiv.org/abs/1802.05799.

Shekel, J. (1971). Test functions for multimodal search techniques. *Fifth Annual Princeton Conf. on Information Science and Systems, 1971*.

Sirovich, L. (1987). Turbulence and the dynamics of coherent structures. I. Coherent structures. *Quarterly of applied mathematics*, 45(3), 561–571.

Snoek, J., Ovadia, Y., Fertig, E., Lakshminarayanan, B., Nowozin, S., Sculley, D., Dillon, J., Ren, J. & Nado, Z. (2019). Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. *Advances in Neural Information Processing Systems*, pp. 13969–13980.

Stokes, J. M., Yang, K., Swanson, K., Jin, W., Cubillos-Ruiz, A., Donghia, N. M., MacNair, C. R., French, S., Carfrae, L. A., Bloom-Ackerman, Z., Tran, V. M., Chiappino-Pepe, A., Badran, A. H., Andrews, I. W., Chory, E. J., Church, G. M., Brown, E. D., Jaakkola, T. S., Barzilay, R. & Collins, J. J. (2020). A Deep Learning Approach to Antibiotic Discovery. *Cell*, 180(4), 688–702.e13. doi: 10.1016/j.cell.2020.01.021.

Sun, X., Pan, X. & Choi, J.-I. (2019). A non-intrusive reduced-order modeling method using polynomial chaos expansion. Consulted at http://arxiv.org/abs/1903.10202.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. & Fergus, R. (2014). Intriguing properties of neural networks. *International Conference on Learning Representations*. Consulted at http://arxiv.org/abs/1312.6199.

Szegedy, C., Ioffe, S., Vanhoucke, V. & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Tao, J. & Sun, G. (2019). Application of deep learning based multi-fidelity surrogate model to robust aerodynamic design optimization. *Aerospace Science and Technology*, 92, 722–737. doi: 10.1016/J.AST.2019.07.002.

Toro, E. F. (2001). *Shock-capturing methods for free-surface shallow flows*. John Wiley.

Valdenegro-Toro, M. (2019). Deep Sub-Ensembles for Fast Uncertainty Estimation in Image Classification. (NeurIPS). Consulted at http://arxiv.org/abs/1910.08168.

Verleysen, M. & François, D. (2005). The Curse of Dimensionality in Data Mining and Time Series Prediction. *Proceedings of the 8th International Conference on Artificial Neural Networks: Computational Intelligence and Bioinspired Systems*, (IWANN'05), 758–770. doi: 10.1007/11494669_93.

Wang, Q., Hesthaven, J. S. & Ray, D. (2019). Non-intrusive reduced order modeling of unsteady flows using artificial neural networks with application to a combustion problem. *Journal of Computational Physics*, 384, 289–307. doi: 10.1016/J.JCP.2019.01.031.

Wu, C., Huang, G. & Zheng, Y. (1999). Theoretical solution of dam-break shock wave. *Journal of Hydraulic Engineering*, 125(11), 1210–1214. doi: 10.1061/(asce)0733-

9429(1999)125:11(1210).

Yang, L., Meng, X. & Karniadakis, G. E. (2020). B-PINNs: Bayesian Physics-Informed Neural Networks for Forward and Inverse PDE Problems with Noisy Data. Consulted at http://arxiv.org/abs/2003.06097.

Yang, Y. & Perdikaris, P. (2019). Adversarial uncertainty quantification in physics-informed neural networks. *Journal of Computational Physics*, 394, 136–152. doi: 10.1016/J.JCP.2019.05.027.

Zhu, Y. & Zabaras, N. (2018). Bayesian deep convolutional encoder–decoder networks for surrogate modeling and uncertainty quantification. *Journal of Computational Physics*. doi: 10.1016/j.jcp.2018.04.018.

Zokagoa, J. M. & Soulaimani, A. (2012a). Low-order modelling of shallow water equations for sensitivity analysis using proper orthogonal decomposition. *International Journal of Computational Fluid Dynamics*, 26, 275–295. doi: 10.1080/10618562.2012.715153.

Zokagoa, J. M. & Soulaimani, A. (2012b). A POD-based reduced-order model for free surface shallow water flows over real bathymetries for Monte-Carlo-type applications. *Computer Methods in Applied Mechanics and Engineering*, s 221–222, 1–23. doi: 10.1016/j.cma.2011.11.012.

Zokagoa, J. M. & Soulaimani, A. (2018). A POD-based reduced-order model for uncertainty analyses in shallow water flows. *International Journal of Computational Fluid Dynamics*, 1–15. doi: 10.1080/10618562.2018.1513496.