Combining supervised deep learning and scientific computing : some contributions and application to computational fluid dynamics

Paul Novello

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAX005

Thèse de doctorat

Combining supervised deep learning and scientific computing: some contributions and application to computational fluid dynamics.

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École Polytechnique

École doctorale n°574 École Doctorale de Mathématiques Hadamard (EDMH)
Spécialité de doctorat : Mathématiques appliquées

Thèse présentée et soutenue à Le Barp, le 09/03/2022, par

PAUL NOVELLO

Composition du Jury :

Marc Schoenauer Directeur de recherche, INRIA Saclay, TAU	Président du Jury
Raphaël Loubère Directeur de recherche, Institut de Mathématiques de Bordeaux, UMR 5251	Rapporteur
Clémentine Prieur Professeure, Université Grenoble Alpes, Laboratoire Jean Kuntzmann	Rapportrice
Marc Massot Professeur, Ecole Polytechnique, CMAP	Examinateur
Sébastien Da Veiga HDR, Safran Tech	Examinateur
Rodolphe Le Riche Directeur de recherche, Ecole des Mines de Saint-Etienne, LIMOS	Examinateur
Gaël Poëtte HDR, CEA CESTA	Co-directeur de thèse
David Lugato Directeur de recherche, CEA CESTA	Co-directeur de thèse
Pietro Marco Congedo Directeur de recherche, INRIA Saclay, Ecole Polytechnique, PLATON	Directeur de thèse


DE LA RECHERCHE À L'INDUSTRIE
cea


Inria


ÉCOLE
POLYTECHNIQUE
IP PARIS

Abstract

Recent innovations in mathematics, computer science, and engineering have enabled more and more sophisticated numerical simulations. However, some simulations remain computationally unaffordable, even for the most powerful supercomputers. Lately, machine learning has proven its ability to improve the state-of-the-art in many fields, notoriously computer vision, language understanding, or robotics. This thesis settles in the high-stakes emerging field of Scientific Machine Learning which studies the application of machine learning to scientific computing. More specifically, we consider the use of deep learning to accelerate numerical simulations.

We focus on approximating some components of Partial Differential Equation (PDE) based simulation software by a neural network. This idea boils down to constructing a data set, selecting and training a neural network, and embedding it into the original code, resulting in a hybrid numerical simulation. Although this approach may seem trivial at first glance, the context of numerical simulations comes with several challenges. Since we aim at accelerating codes, the first challenge is to find a trade-off between neural networks' accuracy and execution time. The second challenge stems from the data-driven process of the training, and more specifically, its lack of mathematical guarantees. Hence, we have to ensure that the hybrid simulation software still yields reliable predictions. To tackle these challenges, we thoroughly study each step of the deep learning methodology while considering the aforementioned constraints. By doing so, we emphasize interplays between numerical simulations and machine learning that can benefit each of these fields.

We identify the main steps of the deep learning methodology as the construction of the training data set, the choice of the hyperparameters of the neural network, and its training. For the first step, we leverage the ability to sample training data with the original software to characterize a more efficient training distribution based on the local variation of the function to approximate. We generalize this approach to general machine learning problems by deriving a data weighting methodology called Variance Based Sample Weighting. For the second step, we introduce the use of sensitivity analysis, an approach widely used in scientific computing, to tackle neural network hyperparameter optimization. This approach is based on qualitatively assessing the effect of hyperparameters on the performances of a neural network using Hilbert-Schmidt Independence Criterion. We adapt it to the hyperparameter optimization context and build an interpretable methodology that yields competitive and cost-effective networks. For the third step, we formally define an analogy between the stochastic resolution of PDEs and the optimization process at play when training a neural network. This analogy leads to a PDE-based framework for training neural networks that opens up many possibilities for improving existing optimization algorithms. Finally, we apply these contributions to a computational fluid dynamics simulation coupled with a multi-species chemical equilibrium code. We demonstrate that we can achieve a time factor acceleration of 18.7 with controlled to no degradation from the initial prediction.

Abstract

Les innovations récentes en mathématiques, en informatique et en ingénierie ont permis de réaliser des simulations numériques de plus en plus complexes. Cependant, certaines simulations restent inabordables en termes de temps de calcul, même pour les super calculateurs les plus puissants. Récemment, l'apprentissage automatique a démontré sa capacité à améliorer l'état de l'art dans de nombreux domaines, notamment la vision par ordinateur, la compréhension du langage et la robotique. Cette thèse s'inscrit dans le domaine émergent et à fort enjeu de l'apprentissage automatique scientifique, qui étudie l'application de l'apprentissage automatique au calcul scientifique. Plus précisément, nous nous intéressons à l'utilisation de l'apprentissage profond pour accélérer des simulations numériques.

Pour atteindre cet objectif, nous nous concentrons sur l'approximation de certaines parties des logiciels de simulation basés sur des Equations Différentielles Partielles (EDP) par un réseau de neurones. La méthodologie proposée s'appuie sur la construction d'un ensemble de données, la sélection et l'entraînement d'un réseau de neurones et son intégration dans le logiciel original, donnant lieu à une simulation numérique hybride. Malgré la simplicité apparente de cette approche, le contexte des simulations numériques implique des difficultés spécifiques. Puisque nous visons à accélérer des simulations, le premier enjeu est de trouver un compromis entre la précision des réseaux de neurones et leur temps d'exécution. En effet, l'amélioration de la première implique souvent la dégradation du second. L'absence de garantie mathématique sur le contrôle de la précision numérique souhaitée inhérent à la conception du réseau de neurones par apprentissage statistique constitue le second enjeu. Ainsi nous souhaiterions maitriser la fiabilité des prédictions issues de notre logiciel de simulation hybride. Afin de satisfaire ces enjeux, nous étudions en détail chaque étape de la méthodologie d'apprentissage profond. Ce faisant, nous mettons en évidence certaines similitudes entre l'apprentissage automatique et la simulation numérique, nous permettant de présenter des contributions ayant un impact sur chacun de ces domaines.

Nous identifions les principales étapes de la méthodologie d'apprentissage profond comme étant la constitution d'un ensemble de données d'entraînement, le choix des hyperparamètres d'un réseau de neurones et son entraînement. Pour la première étape, nous tirons parti de la possibilité d'échantillonner les données d'entraînement à l'aide du logiciel de simulation initial pour caractériser une distribution d'entraînement plus efficace basée sur la variation locale de la fonction à approcher. Nous généralisons cette observation pour permettre son application à des problèmes variés d'apprentissage automatique en construisant une méthodologie de pondération des données appelée "Variance Based Sample Weighting". Dans un deuxième temps, nous proposons l'usage de l'analyse de sensibilité, une approche largement utilisée en calcul scientifique, pour l'optimisation des hyperparamètres des réseaux de neurones. Cette approche repose sur l'évaluation qualitative de l'effet des hyperparamètres sur les performances d'un réseau de neurones à l'aide du critère d'indépendance de Hilbert-Schmidt. Les adaptations au contexte de l'optimisation des hyperparamètres conduisent à une méthodologie interprétable permettant de construire des réseaux de neurones à la fois performants et précis. Pour la

troisième étape, nous définissons formellement une analogie entre la résolution stochastique d'EDPs et le processus d'optimisation en jeu lors de l'entrainement d'un réseau de neurones. Cette analogie permet d'obtenir un cadre pour l'entraînement des réseaux de neurones basé sur la théorie des EDPs, qui ouvre de nombreuses possibilités d'améliorations pour les algorithmes d'optimisation existants. Enfin, nous appliquons ces méthodologies à une simulation numérique de dynamique des fluides couplée à un code d'équilibre chimique multi-espèces. Celles-ci nous permettent d'atteindre une accélération d'un facteur 18.7 avec une dégradation de la précision contrôlée ou nulle par rapport à la prédiction initiale.

Remerciements

Avant toute chose, je tiens à remercier chaleureusement les membres du jury de cette thèse: Marc Schoenauer; Marc Massot; Rodolphe Le Riche; Sébastien Da Veiga; Raphaël Loubere et Clémentine Prieur, pour leur attention ainsi que pour leurs remarques ayant contribué à une discussion scientifique particulièrement enrichissante lors de la soutenance. Merci à Raphaël Loubère et à Clémentine Prieur d'avoir accepté le rôle de rapporteur. Les rapports de thèse m'ont beaucoup apporté, que cela soit à travers les critiques - m'ayant transmis un précieux recul sur mes travaux ainsi que de nombreuses perspectives d'améliorations - ou les encouragements - toujours agréables à recevoir.

A l'origine de cette thèse... il y avait David. Merci d'avoir monté ce projet, de m'y avoir inclus, et d'avoir constitué cette petite équipe en faisant signe à Gaël et Pietro. Merci pour ton implication, tes conseils et ces innombrables discussions dans ton bureau - pas forcément très scientifiques. Gaël, ce fut un plaisir de naviguer avec toi entre bières et théorie de l'approximation. Plus sérieusement, je ne cesserai jamais de me sentir chanceux d'avoir été encadré par quelqu'un d'aussi accessible, bienveillant, passionné, tout en étant si rigoureux, affuté, encyclopédique... Parfois à l'excès (pour chacune de ces qualités) ! Merci Pietro d'avoir apporté ta vision d'ensemble, ton esprit de synthèse et ton expérience. Je suis honoré d'avoir pu travailler avec toi. Passer trois années avec trois encadrants si impliqués fut assurément très riche. Cela fut également à la fois fatiguant : à quatre, il est rare d'être toujours du même avis; et réconfortant : il y avait souvent quelqu'un pour être d'accord avec moi !

Merci aux collègues du Cesta pour leur bonne humeur, à la bande de thésards et postdocs pour les discussions sur fond de bruit de machine à café en fin de vie. Et pour certains, aux bien-aimés et regrettés Umami et Sur-Me. Je regrette également que le covid m'ait empêché d'aller plus régulièrement à l'Inria. Le peu de temps que j'y ai passé avec mes comparses "Platoniciens" m'a toujours donné envie... d'en passer encore plus.

Ces trois dernières années furent professionnellement bien remplies. J'ai cependant la chance d'avoir beaucoup gagné par ailleurs. J'ai gagné une femme, Marie, que je suis comblé d'avoir épousé au beau milieu de cette thèse. Mon mariage a officialisé mon entrée dans une nouvelle famille à laquelle je suis fier d'appartenir. J'ai également gagné par ce mariage ma bande de témoins, que je garderai toute ma vie. J'ai gagné un acolyte qui en aurait assurément fait partie si je l'avais rencontré plus tôt. J'ai gagné une nouvelle bande de potes Niço-Toulousains. Pour la montagne, je ne peux pas en dire autant : c'est elle qui m'a gagné.

Je dédie enfin cette thèse à mes racines, ceux qui étaient là dès le début: ma famille. Novello et Moulonguet se reconnaîtront. J'adresse une marque d'affection particulière à mes grands-parents, et surtout à Papa, Maman, Delphine, Martin (et Pirouette). Je vous dois ce que je suis aujourd'hui et ce que je serai demain.

Contents

Notations

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets

$\mathcal{S}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathcal{A} \backslash \mathcal{B}$	Set subtraction, i.e., the set containing the elements of $\mathcal{A}$ that are not in $\mathcal{B}$
$\mathcal{C}^p$	The set of p times differentiable functions.

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
a_i	Element i of the random vector $\mathbf{a}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) or G(f)(\boldsymbol{x})$	The Jacobian matrix of f at input point $\boldsymbol{x}$
$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathcal{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathcal{S}$

Probability and Information Theory

$p_{\mathrm{x}}(x)$	The probability density function (pdf) of a variable x
$\mathbb{P}_{\mathrm{x}}$	The probability distribution of a variable x
$\mathbb{P}(a)$	The probability of an event a
$\mathrm{x} \sim p_{\mathrm{x}}$	Random variable a has pdf p_{x}
$\mathrm{x} \sim \mathbb{P}_{\mathrm{x}}$	Random variable a follows distribution $\mathbb{P}_{\mathrm{x}}$
$d\mathbb{P}_{\mathrm{x}}(x)$	The probability measure of $\mathbf{x}$ corresponding to $p_{\mathrm{x}}(x)dx$ if $\mathbf{x}$ is continuous, and $\sum \delta_{\boldsymbol{x}_i}(\boldsymbol{x})$ if $\mathbf{x}$ is discrete with values $\{\boldsymbol{x}_i\}$, $i \in \mathcal{S} \subseteq \mathbb{N}$
$\mathbb{E}[f(\mathrm{x})]$	Expectation of $f(x)$ with respect to $p(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathcal{A} \to \mathcal{B}$	The function f with domain $\mathcal{A}$ and range $\mathcal{B}$
$f \circ g$	Composition of the functions f and g
f_θ	A function parametrized by θ.
$\log x$	Natural logarithm of x
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
$\mathbf{1}_{\mathrm{condition}}$	is 1 if the condition is true, 0 otherwise

Chapter 1

Introduction

In the collective imagination, simulations are closely related to Artificial Intelligence (AI). According to some theories (more or less credible), we may live in a simulation[1], either created by machines or other beings, mimicking the world and human intelligence. Some even state that "the answer to life, the universe, and everything" can be represented with 6 bits [2].

There is still a long way to go before these theories can be validated or refuted. Indeed, although innovations in mathematics, computer science, and engineering have enabled more and more complicated numerical simulations, computational experiments conducted by research and industry are still limited by computing capabilities. Indeed, even the largest supercomputers struggle to simulate physics at its most refined scale. Besides, even though AI has known countless breakthroughs in the last decades, we are still far from general artificial intelligence replicating human intelligence.

On the one hand, scientific computing and numerical analysis have been developed to solve coupled systems of Partial Differential Equations (PDE) that model a phenomenon of interest, with applications, for instance, in computational physics, biology, economy, and climatology. On the other hand, AI, or machine learning, has been notorious for its tremendous computer vision, robotics, and natural language understanding achievements. Scientific computing and AI have long been poled apart, but recently, interest has grown to combine them to unlock new advances. Their cross-fertilization has fostered the emergence of a new field called scientific machine learning (Baker et al., 2019). This thesis is part of this new field and aspires to contribute to this cross-fertilization.

[1] Simulacron 3 (Galouye, 1964), The Matrix (Wachovski, 1999), Are you living in a computer simulation? (Bostrom, 2003), L'Anomalie (Le Tellier, 2020)

[2] The Hitchhiker's Guide to the Galaxy (Adams, 1978)

1.1 Challenges and stakes of numerical simulations

Numerical simulations and scientific computing are ubiquitous in research and industry. In research, it allows conducting computational experiments to validate or explore knowledge about natural sciences. In industry, it is at the basis of complex system conception. However, despite the immense computational power of modern supercomputers, numerical simulations always face a **performance-accuracy trade-off**.

Indeed, to be accurate and reliable, a numerical simulation has to model accurate physics. Nonetheless, in that case, the code execution can be lengthy, computationally intensive, and sometimes even not affordable. Therefore, when working on a numerical simulation code, scientists and engineers have to find the right level of accuracy to reliably simulate a phenomenon, together with the right amount of simplifications to make it workable.

That is why there are high stakes at accelerating numerical simulation codes: it would induce valuable time savings for single simulation runs that are of interest for forecasting, as well as for parametric studies like uncertainty quantification, sensitivity analysis, or calibration intensively used for system conception and decision-making. Finally, accelerating simulation codes may also enable currently unavailable simulations.

1.2 Supervised deep learning as a way to speed-up simulation codes

Many coupled systems of equations, solved in simulation codes, can be decomposed into, for example, two operators:

$$\begin{cases} F_1(\boldsymbol{\eta}, \boldsymbol{u}) = 0, & \text{(1.1a)} \\ F_2(\boldsymbol{u}, \boldsymbol{\eta}) = 0, & \text{(1.1b)} \end{cases}$$

where F_1 and F_2 are mathematical operators acting on the set of unknowns $(\boldsymbol{u}, \boldsymbol{\eta})$. Depending on the physics of interest, the operators can strongly differ. For instance, in burn-up applications (neutronics) (Bernède and Poëtte, 2018; Dufek et al., 2013), F_1 is the linear Boltzmann equation of unknown the density of neutrons $\boldsymbol{u}$. For this equation, $\boldsymbol{\eta}$ is only a parameter. On the other hand, $\boldsymbol{\eta}$ the vector of isotopic densities is the unknown of F_2, the Bateman system. In radiation hydrodynamics (Mihalas and Mihalas, 1984; Castor, 2004; Clouët and Samba, 2004), F_1 is the Euler system of unknown $\boldsymbol{u}$, the vector made of the mass density, the momentum and of the total energy, and F_2 is the Boltzmann equation of unknown $\boldsymbol{\eta}$, the density of energy of photons. In Computational Fluid Dynamic (CFD) for non-viscous fluids (Maire et al., 2007b; Loubère et al., 2014), F_1 is the Euler system of unknown $\boldsymbol{u}$, the vector made of the mass density, the momentum and of the total energy, and F_2 is simply the call for the equation of state relating the main variable $\boldsymbol{u}$ to the vector of thermodynamic quantities $\boldsymbol{\eta}$ (pressure, temperature, sound speed). Independently of

the physics of interest, the iterative resolution of the coupled system often needs successive resolutions of equation (1.1a) and equation (1.1b) within an iteration. It is typically the case for operator splitting (Strang, 1968). Hence, even if the resolution of equation (1.1a) alone is not costly, the resolution of equation (1.1a) coupled with equation (1.1b) might be. In this case, most of the computational time would be spent on the resolution of equation (1.1b).

There are already several ways to tackle this performance-accuracy trade-off. The most classical one consists of making simplifying physical assumptions. Assume that $\boldsymbol{\eta}$ does not vary that much when equation (1.1b) is solved. Then, solving equation (1.1a) could be enough to satisfy accuracy constraints and allow a cost-effective resolution when equation (1.1b) is neglected. One finer possibility is to construct abacuses from the resolution of equation (1.1b) before the execution and then interpolate in these abacuses at run time instead of solving equation (1.1b) (Sigrist, 2019, 2020). This is particularly used when equation (1.1b) is an equation of state. Another popular method, called surrogate modeling, trains a surrogate model to learn a mapping from input parameters to output quantities of interest. The numerical simulation - here, the coupling between equation (1.1a) and equation (1.1b) - is considered as a black-box function used to build the training data (Gramacy, 2020). Then, the surrogate model is used in place of the simulation for parametric studies. However, all these methods have their flaws:

- Simplifying hypotheses forbids taking into account finer phenomena. In our case, it might be a problem where it is impossible to neglect F_2 reasonably.
- Abacuses are irrelevant in high dimension because the domain of interest may be too large to fill with interpolation points correctly. Moreover, interpolating at run time may become computationally costly, for instance, when $\boldsymbol{\eta}$'s dimensions are too high.
- The whole simulation code being computationally expensive, it may be impossible to construct a sufficient database for black-box surrogate modeling to be effective.

A less common approach for accelerating simulation codes consists of approximating only the resolution of equation (1.1b) with a surrogate model and then plugging it inside the simulation code, yielding a hybrid simulation code (see Kluth et al. (2019); Behler and Parrinello (2007); Han et al. (2019); Stecher et al. (2014) for examples). This approach solves the flaws of the methods previously described:

- There is no simplifying hypothesis, and the phenomenon described by equation (1.1b) is included in the simulation.
- One can choose a model that performs well in high dimensions.
- Even if most of the computational time is spent on the resolution of equation (1.1b), this system is generally cheap to solve when considered independently. Therefore, it is possible to build a large database for fitting a surrogate model. Moreover, this surrogate can be reused for any simulation code that calls for the resolution of equation (1.1b).

In this thesis, we explore this approach and choose to use neural networks as surrogate models. Neural networks are machine learning models that consist of artificial layers-structured neurons connected by weights. They can have different shapes or architectures, characterized by criteria called hyperparameters. The choice of this approach is motivated by several elements.

First, deep learning is one of the main building blocks of AI and is responsible for many of its most eloquent breakthroughs. Furthermore, recent advances have dramatically improved deep learning in terms of software with the research around optimization and neural networks' architectures; and hardware with the exploitation of GPUs and TPUs. Hence, neural networks are more and more accurate and more and more cost-effective.

Second, once trained, a neural network can be easily saved, exported, and used as a simple function to be plugged inside a numerical simulation code. Moreover, its implementation boils down to vector products, which allows processing batch inputs efficiently. Thus, it is particularly suited to numerical simulations conducted on meshes, which are arrays-like structures.

Finally, neural networks are very flexible, and their execution complexity does not depend on the number of points used for training and only depends linearly on their input and output dimensions. To sum up, supervised deep learning is a promising approach to accelerate numerical simulation codes.

1.3 Supervised deep learning methodology in the eye of numerical and uncertainty analysis

The methodology of supervised deep learning to construct a surrogate model based on a neural network can be divided into three steps:

1 The construction of the training database using the code that solves F_2.

2 The choice of the neural network's architecture, based on hyperparameters.

3 The training of the network using optimization algorithms.

When considering this methodology through the prism of the performance-accuracy trade-off, each of its steps becomes important and echoes the challenges and stakes of numerical simulation. Indeed, the construction of the database has a substantial impact on the performance of the reduced model. This impact has justified many works about designs of experiments. The training database is crucial to ensure the good accuracy of the model, and it is also a way to improve its accuracy without affecting its execution time. Then, many achievements of deep learning were due to innovative architectures. That is why hyperparameters are also crucial to obtain a competitive accuracy. Carefully selecting hyperparameters' values is all the more

important since it also impacts the cost-efficiency of the neural networks. In our case, we cannot simply choose them to maximize the accuracy: we have to make sure that the neural network remains cost-effective. Finally, similarly to the construction of the training database, the optimization process of neural networks has a significant impact on its accuracy without affecting its execution time.

In this thesis, motivated by these observations, we investigate each of these steps. Considering the field of application of this methodology, i.e. scientific computing, we take the opportunity to examine these steps in the eye of best practices from numerical and uncertainty analysis. As we shall see, such an approach can benefit both general supervised deep learning and numerical simulations.

1.4 Contributions

This thesis gathers four contributions. The first three focus on each step of building surrogate models with supervised deep learning, and the last contribution is the application of this methodology for the construction of a hybrid simulation code in CFD.

1.4.1 Training distribution and local variation.

We elaborate on the intuition that in approximating a function f, a neural network is more accurate when the training database focuses on regions where f is steeper. We derive an illustrative generalization bound to legitimate this intuition and verify it empirically. To that end, we use Taylor expansion of f to build a sampling scheme for constructing the training database that we call Taylor-based Sampling (TBS). TBS proves to be promising but relies on too stringent hypotheses for large-scale numerical simulations and deep learning in general. Indeed, using Taylor expansion of f implies having access to its derivatives, which is often not the case in practice, either because they are too costly to obtain or not defined. Moreover, adding new sampled points to the training set requires evaluating f on these points, which is impossible in general machine learning because we do not have access to f. For these reasons, we construct a weighting scheme that alleviates these hypotheses and improves neural network accuracy on various machine learning tasks.

Presentations

- A Taylor Based Sampling Scheme for Machine Learning in Computational Physics *Paul Novello, Gael Poette, David Lugato and Pietro Congedo, Second Workshop on Machine Learning and the Physical Sciences (NeurIPS 2019), Vancouver, Canada.*
- A Taylor Based Sampling Scheme for Deep Learning in Computational Physics: L'avantage du deep learning pour les simulations numériques.

Paul Novello, Gael Poette, David Lugato and Pietro Congedo, Machine Learning Meetup, 2019, Pau, France

Publications

- Leveraging Local Variation in Data: sampling and weighting schemes for supervised deep learning
  Paul Novello, Gael Poette, David Lugato and Pietro Congedo, Submitted to the Journal of Machine Learning for Modeling and Computing

1.4.2 Hyperparameter optimization using goal-oriented sensitivity analysis.

It is commonly accepted that hyperparameters have a strong impact on the accuracy of neural networks. Another observation is that it is also difficult to characterize this impact (Bergstra and Bengio, 2012): from one problem to the other, the same hyperparameter can be more or less impactive. This work introduces a thorough goal-oriented sensitivity analysis of the neural network's accuracy concerning its hyperparameters, based on the Hilbert-Schmidt Independence Criterion (HSIC). We adapt HSIC to hyperparameter spaces, which is challenging because hyperparameters can be discrete (width of the neural network), continuous (learning rate), categorical (activation function), or boolean (batch normalization (Ioffe and Szegedy, 2015)); some hyperparameter's presence is conditional to others (e.g. moments decay rates specific to Adam optimizer (Kingma and Ba, 2015)); and they can strongly interact (as shown in Tan and Le (2019), in some cases, it is better to increase depth and width by a similar factor). The obtained metric allows building hyperparameter optimization methodologies that are interpretable and yields competitive and cost-effective neural networks.

Talks

- Explainable Hyperparameters Optimization using Hilbert-Schmidt Independence Criterion
  Paul Novello, Gael Poette, David Lugato and Pietro Congedo, MASCOT 2021 Meeting, gdr-mascotnum, virtual

Publications

- Goal-Oriented Sensitivity Analysis of Hyperparameters in Deep Learning
  Paul Novello, Gael Poette, David Lugato and Pietro Congedo, Submitted to the Journal of Scientific Computing.

1.4.3 A view of learning from the Partial Differential Equation theory

We build an analogy between the resolution of PDE and the training of neural networks, leading to a PDE framework for learning. Based on this framework, we enhance classical stochastic gradient descent with new terms that speed up the convergence in convex regions of the parameter spaces from our experiments. We also use a stability condition from PDE resolution theory that translates into constraints on the learning rate, which allows efficiently exploring the parameter space while maintaining the stability of the process without any learning rate tuning. These improvements are showcased on a simple two-dimensional optimization problem involving training a neural network with two neurons.

1.4.4 Efficient hybrid numerical simulations (with guarantees)

We apply the neural network-based acceleration approach described in Section 1.2 to the simulation of a hypersonic flow around an object during atmospheric reentry. To accurately simulate this phenomenon, it is necessary to compute the chemical equilibrium of species found in the fluid. We approximate the chemical reactions with a neural network and obtain a significant speed-up for a comparable accuracy. Using error analysis based on uncertainty propagation, we show that the hybrid code's error is negligible compared to other sources of errors that classical numerical simulation codes usually undergo. We also describe how to obtain the exact predictions with the hybrid code by using its output to initialize the computations of the original code. This allows reaching a speed-up factor of 10.

Talks

- Accelerating hypersonic reentry simulations using deep learning-based hybridization (with guarantees)
  Paul Novello, Gael Poette, David Lugato and Pietro Congedo, HSA'2021, IRT SystemX, Saclay, France

Publications

- Accelerating hypersonic reentry simulations using deep learning-based hybridization (with guarantees)
  Paul Novello, Gael Poette, David Lugato and Pietro Congedo
  In preparation. To be sent to the Journal of Computational Physics

1.5 Thesis organization

In Chapter 2, we introduce some basic notions about neural networks and supervised deep learning that we organize around the previously described three-step methodology. Using a decomposition of the neural network output error, we emphasize the importance of each of these steps, to which the following three chapters are dedicated.

Chapter 3 focuses on the link between local variation in training data and learning difficulty, chapter 4 introduces the use of HSIC based goal-oriented sensitivity analysis for hyperparameter optimization, and chapter 5 investigates the link between learning and PDE theory.

Finally, Chapter 6 is dedicated to the application of this methodology to the acceleration of the fluid dynamics simulation code.

To emphasize the parallel between the thesis structure and the supervised deep learning methodology and to help the reader visualize its progression throughout the manuscript, we provide a summary diagram in Figure 1.1. We clip it at the beginning of each chapter (excepted Chapter 2).

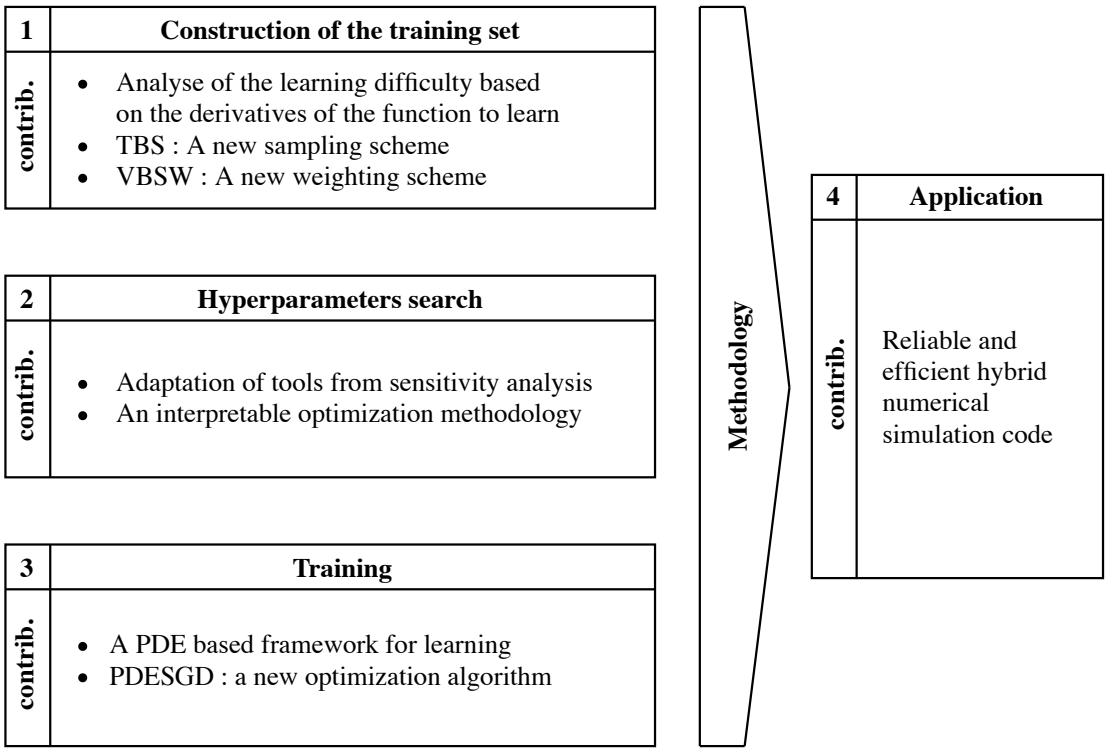


Figure 1.1: Methodology for supervised deep learning in numerical simulations and contributions of the thesis

Chapter 2

Supervised deep learning and numerical simulations

In this chapter, we introduce basic notions that we use throughout this thesis. First, we formalize the task of supervised machine learning as a statistical estimation and optimization problem. Then, we describe neural networks that form the basis of deep learning. We emphasize them as parametric models whose mathematical definition can be explicit. It clarifies the status of neural networks and gets them closer to classical statistical models because it is formulated as a model whose parameters have to be tuned based on data. We also highlight the importance of prior knowledge, or inductive bias, for constructing diverse neural networks architectures like convolutional neural networks or recurrent neural networks. Finally, we detail the challenges and stakes of the training process of neural networks, which is a non-convex optimization problem.

As mentioned in Section 1.3, these three points, namely the formulation of the supervised learning task, the architecture of neural networks, and their optimization, can be put together as steps in the methodology of supervised deep learning. The first step consists of defining the estimation problem and the training data, the second of finding hyperparameters that define the architecture, and the third of training the neural network using optimization algorithms. The importance of each step for deep learning and numerical simulation is highlighted in the last sections, through discussions on the loss function and the specificities of applying machine learning to numerical simulation.

Contents

2.1 Supervised machine learning

In the last decade, machine learning has flourished to become a major field in research and industry. This breakthrough is strongly related to the massive increase in data availability and to the rise of deep learning. Indeed, many applications that highlight machine learning capabilities, such as image processing or language understanding, rely on the profusion of data. Most of these applications are based on what is called a supervised learning approach. Other successful approaches are dedicated to learning when data availability is limited, such as unsupervised learning, semi-supervised learning, or few-shot learning. Moreover, some tasks only make use of simulated or dynamically generated data, such as reinforcement learning. Nonetheless, most of the time, the learning process contains steps formulated as supervised learning. In this section, we formally describe this task.

Supervised machine learning consists of learning a mapping between input and output data, often called features and targets. To that end, a model is constructed and optimized to fit this mapping best.

More specifically, we can denote the mapping to learn as a function $f : \mathbf{S} \subset \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}}$ where $\mathbf{S}$ is a measured sub-space of $\mathbb{R}^{n_{in}}$ depending on the considered application. Then, learning comes to approximate f with a machine learning model. We denote such a model by $f_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} \in \Theta$ contains all the parameters of the model. Optimizing $f_{\boldsymbol{\theta}}$ to approximate f means finding a value $\boldsymbol{\theta}^*$ for $\boldsymbol{\theta}$ that minimizes an integrated loss function $J_{\mathbf{x}}(\boldsymbol{\theta}) = \mathbb{E}[L(f_{\boldsymbol{\theta}}(\mathbf{x}), f(\mathbf{x}))] = \int_{\mathbf{S}} L(f_{\boldsymbol{\theta}}(\boldsymbol{x}), f(\boldsymbol{x}))\mathbb{P}_{\mathbf{x}}$, where L is a loss function, $L : \mathbb{R}^{n_{out}} \times \mathbb{R}^{n_{out}} \to \mathbb{R}$. The random variable $\mathbf{x} \sim \mathbb{P}_{\mathbf{x}}$ models the distribution of the input data in $\mathbf{S}$.

In practice, we may not have access to $\mathbb{P}_{\mathbf{x}}$ and in order to optimize $f_{\boldsymbol{\theta}}$, $J_{\mathbf{x}}$ is estimated by $\widehat{J}_{\mathbf{x}}$ using N points $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\} \in \mathbf{S}$ drawn from $\mathbf{x}$, and their point-wise values $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$.

These sets are the input and output data, or features and targets. The supervised learning task can be summarized as :

The supervised learning task

Find $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{argmin}}\, J_{\mathbf{x}}(\boldsymbol{\theta})$ by minimizing

$$\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \alpha_i L(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), f(\boldsymbol{x}_i)) \approx \mathbb{E}[L(f_{\boldsymbol{\theta}}(\mathbf{x}), f(\mathbf{x}))], \tag{2.1}$$

with $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\} \in \mathbf{S}$ drawn from $\mathbf{x}$, $\{\alpha_1, ..., \alpha_N\} \in \mathbb{R}^{+N}$ estimation weights such that $\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta}) \underset{N\to\infty}{\to} J_{\mathbf{x}}(\boldsymbol{\theta})$.

At this point, we have provided a formulation of supervised learning that only implies data $\{(\boldsymbol{x}_1, f_{\boldsymbol{\theta}}(\boldsymbol{x}_1)), ..., (\boldsymbol{x}_N, f_{\boldsymbol{\theta}}(\boldsymbol{x}_N))\}$ and any machine learning model $f_{\boldsymbol{\theta}}$.

Remark. *This framework encompasses all parametric models constructed from data, including polynomial interpolation, linear and kernel regression, kriging, and is not specific to neural networks. Besides, even if the keyword "machine learning" has been popularized only recently, it refers to a very classical task that dates back to the method of least squares, introduced by Legendre in 1805.*

2.2 Artificial Neural Networks

Neural networks are machine learning models that are the main building block of deep learning. It has been first introduced with an electronic hardware implementation by McCulloch and Pitts (1943), and with a software implementation by Widrow and Hoff (1960). However, its use has only become widespread since the recent breakthrough of deep learning in image classification during the 2012 edition of Imagenet Large Scale Visual Recognition Challenge (ILSVRC). On that occasion, a neural network called AlexNet (Krizhevsky et al., 2012) dominated the leaderboard and broke the previous record by achieving a 10.8 percentage points lower classification error. In this section, we mathematically define neural networks. We also discuss the meaning of architectures in deep learning, and we give an overview of the fundamental architectures that form the basis of deep learning.

2.2.1 Definition

We mentioned a parametric machine learning model $f_{\boldsymbol{\theta}}$ defined by its parameters $\boldsymbol{\theta}$. When the model is a neural network, choosing $\boldsymbol{\theta}$ alone is not enough to define it completely. Indeed, a neural network can be represented by a succession of interconnected layers of neurons, also

called units, starting with an input layer and ending with an output layer. We talk about deep neural networks when there is more than one hidden layer. The way these layers are interconnected has an impact on the execution of the neural network. We call architecture a specific type of interconnection. There may be many ways of arranging these interconnections, which explains why the very furnished literature on deep learning is crowded with many different neural networks names. To begin with, let us describe the most simple form of neural networks: fully connected neural networks, also called Multi-Layer Perceptron (MLP).

A MLP can be fully described by two sets of parameters. The first set of parameters describes its shape, or architecture :

- d the depth of the network, i.e. the number of hidden layers,
- n_k the number of neurons in the network's k-th layer, with $k \in \{0, d+1\}$ (with $n_{d+1} = n_{out}$ and $n_0 = n_{in}$),
- σ_k the activation function of the k-th layer, generally non linear.

These parameters belong to a set of parameters called hyperparameters, which describe the neural network's architecture and its learning framework. The second set of parameters, corresponding to $\boldsymbol{\theta}$, contains the parameters to be optimized during the learning process :

- $\boldsymbol{W}^k = \{\omega_{ij}^k\}, (i,j) \in \{0, n_k - 1\} \times \{0, n_{k-1} - 1\}$ the weights matrix between $(k-1)$-th and k-th layers,
- $\mathbf{b}^k = (b_0, ..., b_{n_k - 1})$ the bias vector of the k-th layer.

Let us define $f^k : \mathbb{R}^{n_k} \to \mathbb{R}^{n_{k+1}}$ such that

$$f_i^k(\boldsymbol{x}) = \sigma_k\Big(\sum_{j=0}^{j=n_k-1} \omega_{ij}^k x_j + b_i^k\Big) = \sigma_k(\boldsymbol{W}^k \boldsymbol{x} + \boldsymbol{b}^k) \tag{2.2}$$

with $\boldsymbol{x} = (x_1, ..., x_{n_k}) \in \mathbb{R}^{n_k}$. Then, $f_{\boldsymbol{\theta}} : \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}}$, the function of the network can be written

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f^{d+1} \circ \cdots \circ f^1(\boldsymbol{x}) \tag{2.3}$$

with $\boldsymbol{\theta} = \{\boldsymbol{W}^k, b_k | k \in \{1, d+1\}\}$. Equation (2.2) and equation (2.3) emphasize that the execution of a neural network, called forward pass, can be seen as a succession of matrix vector products composed with activation functions. In Figure 2.1, we illustrate a MLP of depth $d = 1$ (there is one hidden layer), $n_1 = 3$ (the hidden layer has 3 neurons), with $n_{in} = 2$

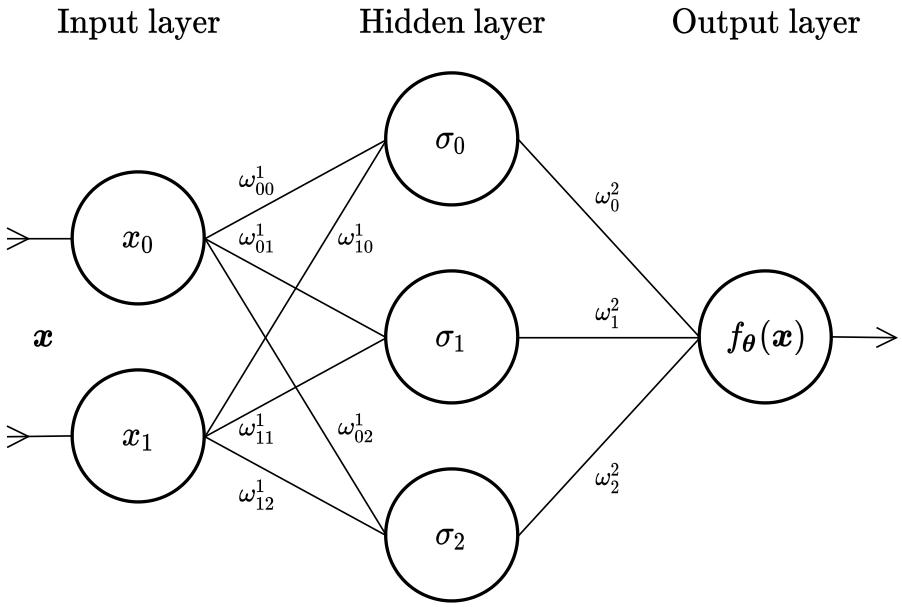


Figure 2.1: MLP of depth $d = 1$, with $n_{in} = n_0 = 2$, $n_1 = 3$ and $n_{out} = n_2 = 1$. In this specific case, for readability, we denote by σ_i the i-th component of the output of the hidden layer.

(the input dimension is 2) and $n_{out} = 1$ (the output dimension is 1). For this network, we have

$$\begin{cases} f_{\boldsymbol{\theta}}(\boldsymbol{x}) & = \omega_0^2\sigma(\omega_{00}^1 x_0 + \omega_{10}^1 x_1 + b_0^1) + \omega_1^2\sigma(\omega_{01}^1 x_0 + \omega_{11}^1 x_1 + b_1^1) + \omega_2^2\sigma(\omega_{02}^1 x_0 + \omega_{12}^1 x_1 + b_2^1) + b^2 \\ & = \sum_{i=0}^{2} \omega_i^2 \sigma(\sum_{j=0}^{1} \omega_{ji}^1 x_j + b_i) + b^2 \\ \boldsymbol{\theta} & = \{\omega_0^2, \omega_{00}^1, \omega_{10}^1, \omega_1^2, \omega_{01}^1, \omega_{11}^1, \omega_2^2, \omega_{02}^1, \omega_{12}^1, b_0^1, b_1^1, b_2^1, b^2\}. \end{cases}$$

With this formulation, neural networks can be apprehended from the point of view of approximation theory. Indeed, once the architecture is chosen, neural networks can be seen as a parametric class of function approximators. Hornik et al. (1989) first demonstrates a universal approximation theorem for neural networks with $d = 1$ and arbitrary width. It makes deep learning close to polynomial regression, which benefits from the Stone-Weierstrass theorem. Later on, Barron (1994) gives an upper bound for the L_2 error of a neural network approximating a function, with respect to the number of parameters of the model, and Lu et al. (2017) extends the universal approximation theorem to networks of width $n_{in} + 4$ with arbitrary depth. The analogy with classical approximation theory goes beyond these similarities. Indeed, the last layer of a neural network is often constructed as a classical general linear regression, with σ_{d+1} chosen as the identity or a logistic function. Therefore, we can understand deep learning as a process that constructs a basis, whose span is called feature space, where the input data is projected, and linear regression is performed. This process is precisely the same as polynomial regression, where the basis is arbitrarily chosen to

be polynomials, or gaussian process regression, shown to be a linear regression conducted on vectors from a Restricted Kernel Hilbert Space (Rasmussen and Williams, 2005).

Still, very often, neural networks are presented as block diagrams, and a formula such as equation (2.3) is not specified. Indeed, for complex architectures, which are ubiquitous in modern deep learning, the formula may be very tedious to write and to read. In these cases, diagrams provide a quick view of the architecture's structure and are more efficient in conveying the ideas behind its construction. However, as we shall see, many architectures boil down to a particular case of a fully connected neural network.

2.2.2 Architectures as priors on data structure

As opposed to other machine learning models, neural networks are notorious for their profusion of hyperparameters and different architectures. There are almost as many architectures as deep learning applications. In this part, we explain how these architectures are linked to the original MLP. In order to emphasize the link between modern neural networks and MLP, let us first focus on one of the most famous declinations of neural networks: Convolutional Neural networks (CNN). Introduced by Fukushima (1980), the idea of CNN is to adapt neural networks to the specific structure of image data. To that end, instead of processing layers with classical matrix multiplication of full weight matrices, CNN performs convolution operations. Such operations are extensively used in classical image processing. That is why enforcing this *a priori* within the neural network structure makes them very efficient for machine learning tasks based on image data.

Let $\boldsymbol{X} \in \mathbb{R}^n \times \mathbb{R}^n$. The CNN convolution operation involves a kernel $\boldsymbol{K} \in \mathbb{R}^{n_K} \times \mathbb{R}^{n_K}$. When applied to $\boldsymbol{X}$, it outputs another matrix $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{K} \in \mathbb{R}^n \times \mathbb{R}^n$ such that

$$Y_{i,j} = [\boldsymbol{X} * \boldsymbol{K}]_{i,j} = \sum_p \sum_q X(p,q)K(p-i, q-j), \tag{2.4}$$

where $p, q \in \{1, ..., n\}$, and $K : \{-n, ..., n\}^2 \to \mathbb{R}; (i,j) \to K_{i,j}$ and $X : \{-n, ..., n\}^2 \to \mathbb{R}; (i,j) \to X_{i,j}$. Hence, a CNN applied on image data is a succession of matrices processed by convolution operations.

Remark. *The definition of CNN convolution is slightly different from that of the classical convolution, where $K(i-p, j-q)$ is considered instead of $K(p-i, q-j)$.*

To emphasize the link between CNN and MLP, let us write the convolution operation for vectors. Let $\boldsymbol{x} \in \mathbb{R}^{n_{in}}$ and $\boldsymbol{k} \in \mathbb{R}^{n_k}$. Then,

$$y_i = [\boldsymbol{x} * \boldsymbol{k}]_i = \sum_p x(p)k(p-i), \tag{2.5}$$

with x and k defined similarly to X and K. We can write $\boldsymbol{y}$ such that $\boldsymbol{y} = \boldsymbol{W}^*\boldsymbol{x}$, with

$$\boldsymbol{W}^* = \begin{pmatrix} k(n-1) & k(n-2) & ... & k(1) & k(0) \\ k(n-2) & k(n-3) & ... & k(0) & k(-1) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ k(1) & k(0) & ... & k(-n+2) & k(-n+1) \\ k(0) & k(-1) & ... & k(-n+1) & k(-n) \end{pmatrix}.$$

This expression shows that a convolution operation can be written in terms of matrix multiplication, falling back under the MLP framework. It is also possible to reproduce this formulation for convolution operations on matrices by constructing a vector $\boldsymbol{x} \in \mathbb{R}^{n \times n}$ out of the elements of $\boldsymbol{X} \in \mathbb{R}^n \times \mathbb{R}^n$. In practice, a size s is chosen for the kernel k, so that $s = 2n_k + 1$ and $k : \{-n_k, ..., n_k\}$ with $n_k < n$. For example, if $s = 3$,

$$\boldsymbol{W}^* = \begin{pmatrix} 0 & 0 & 0 & ... & 0 & k(1) & k(0) \\ 0 & 0 & 0 & ... & k(1) & k(0) & k(-1) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ k(1) & k(0) & k(-1) & ... & 0 & 0 & 0 \\ k(0) & k(-1) & 0 & ... & 0 & 0 & 0 \end{pmatrix}.$$

Now that it is clear that a CNN can be written as an MLP, let us focus on the differences between those two types of networks by comparing $\boldsymbol{W}^*$ with an MLP's full weight matrix. The matrix $\boldsymbol{W}^*$ has two specificities: it is sparse, and it only has s different elements. These characteristics come from constraints imposed by the definition of the convolution operation. Thanks to these constraints, we no longer have n^2 parameters to optimize, but only s, which is often dramatically lower since, usually, $n_k \ll n$. As a result, the CNNs that we can afford are MLPs that are extremely larger than classical ones. We can think of CNN as giant MLPs, allowing for approximating very complex functions, made trainable thanks to priors on the weight matrices. These priors are adapted to images because they consist in enforcing convolution operations. Hence, a CNN's approximation capacities for image data may be that of a giant MLP, but the complexity of its optimization is that of a regular MLP. The effect of imposing constraints in order to guide the neural networks towards more efficient architectures is called inductive bias.

In this part, we have explained that CNNs are nothing more than MLPs with constraints on their parameters. Most of the other popular types of architecture that we describe in the next section can be formulated as MLPs as well, with weights constraints and thereby inductive bias adapted to their scope of application. However, the implementation of neural networks by modern software frameworks, like TensorFlow Abadi et al. (2015) or Pytorch Paszke et al. (2019), as computational graphs of operations, allows avoiding the MLP formulation. It is for the best since the intuition behind the inductive bias of popular architectures is often only

visible when they are represented as graphs or diagrams.

2.2.3 Some famous architectures

In the last part, we have described convolutional neural networks, which are particularly suited to image processing. Another famous archetype of neural networks is Recurrent Neural Networks (RNN) (Hopfield, 1982), which has been introduced for learning on time series. Their use has been popularized for natural language understanding, where sequences of words are considered time series or audio processing. Since then, some other architectures have been created to make RNNs even more efficient. Notably, Long Short-Term Memory (LSTM) networks, introduced by Hochreiter and Schmidhuber (1997), and Gated Recurrent Units Cho et al. (2014), are still popular. Recently, Attention mechanism (Vaswani et al., 2017a) has obtained tremendous success in language understanding, overshadowing a bit the architectures mentioned above.

Other types of networks aim at encoding different data structures. For instance, Bronstein et al. (2017) introduce the use of neural networks on data that can be represented as graphs and Qi et al. (2017) on points clouds. In a more abstract approach, Sabour et al. (2017) construct neural networks, called capsule networks, capable of encoding the position of the different objects in images.

Closer to the domain of scientific computing, some architectures constructed from physical priors have been proposed for scientific machine learning. For instance, Hamiltonian neural networks (Greydanus et al., 2019) are trained under the constraint of respecting conservation laws in a physical phenomenon. In computational biology, Zhang et al. (2018) introduce a neural network preserving the symmetry in inter-atomic potential energy models. For learning PDE-based phenomenon, Raissi et al. (2019) uses a loss function that drives the neural network towards solutions that satisfy physical laws. However, these approaches, as well as the field of scientific machine learning, are still young. Therefore, the state-of-the-art has yet to stabilize to identify a technique as fundamentals as CNN, RNN, or Attention.

This overview is not exhaustive and is restricted to architectures designed for supervised deep learning. Besides, the aim of this section is not to provide a thorough review of all the deep learning techniques, which would be the topic for a whole textbook (see Goodfellow et al. (2016)). The aim is instead to emphasize that all the different names found in the deep learning literature can be seen as classical neural networks subject to hard constraints imposed to satisfy *a priori* knowledge of the learning problem. It justifies the potential of cross-fertilization between machine learning and scientific computing. Indeed, the latter is full of *a prior* knowledge, and there is still much to do to translate this knowledge into machine learning terms.

2.3 Optimization in deep learning

So far, we have formalized the problem of supervised learning and discussed the role of the architecture of neural networks. In this part, we briefly describe the training process at play in deep learning.

The training of neural networks can be formulated as an optimization problem, as seen in Section 2.1. This optimization problem is known to be particularly challenging. Indeed, the compositions and the non-linearities imposed by activation functions make the optimization problem non-convex. Besides, the problem is known to have many local minima - Auer et al. (1996) demonstrate that there may be exponentially many for a neural network with a single neuron. Hence, the optimization algorithms used to solve this problem are usually lengthy, and the training of deep neural networks is very resource-intensive.

On the one hand, this challenge is difficult, and on the other hand, it carries high stakes due to the potential of neural networks for machine learning and the cost of its training. Hence, it has stimulated important research efforts in the last decades. Stochastic gradient descent (Robbins and Monro, 1951) is the first optimization algorithm that was used to that end and remains very popular. It is an iterative algorithm which updates the parameters in the opposite direction of $\nabla_{\boldsymbol{\theta}} \widehat{J}_{\mathbf{x}}(\boldsymbol{\theta})$. Its success stems from its simplicity and its implementation, based on back-propagation (Rumelhart et al., 1986), that efficiently computes $\nabla_{\boldsymbol{\theta}} \widehat{J}_{\mathbf{x}}(\boldsymbol{\theta})$ at each iteration.

Back-propagation and automatic differentiation

The gradient back-propagation algorithm (Rumelhart et al., 1986) is an automatic differentiation algorithm that allows efficiently computing the gradients of a neural network.
Automatic differentiation is based on the chain rule of derivation that states that for a function $g : \mathbb{R}^p \to \mathbb{R}^q$ such that $g = g^n \circ ... \circ g^1$ with $g^i : \boldsymbol{y}_i \in \mathcal{Y}_i \to g^i(\boldsymbol{y}_i) \in \mathcal{Y}_{i+1}$ with $i \in \{1, ..., n\}$ and $\mathcal{Y}_1 = \mathbb{R}^p$, $\mathcal{Y}_{n+1} = \mathbb{R}^q$,

$$\nabla_{\boldsymbol{y}} g(\boldsymbol{y}) = \nabla_{\boldsymbol{y}_n} g^n \times ... \times \nabla_{\boldsymbol{y}_1} g^1(\boldsymbol{y})$$

It is then possible to compute $\nabla_{\boldsymbol{y}} g(\boldsymbol{y})$ using the product of derivatives of g^i. The back-propagation algorithm is a special case of automatic differentiation. Its name stems from the fact that we compute the gradient of $\hat{J}_{\mathbf{x}}$ with respect to each parameter sequentially, starting from the parameters involved in the last layer of the neural network.

Let us first focus on the last layer of the neural network, with weight matrix $\boldsymbol{W}^d = \{\omega_{ij}^d\}$, $i, j \in \{1, ..., n_d\}^2$. For simplicity, we denote $f^d \circ ... \circ f^1(\boldsymbol{x})$ by $f^d(\boldsymbol{x})$. We also consider $\nabla_{\boldsymbol{\theta}} L$ rather than $\nabla_{\boldsymbol{\theta}} \hat{J}_{\mathbf{x}}(\boldsymbol{\theta})$, which does not change the algorithm. We have:

$$\frac{\partial L}{\partial \omega_j^d}(\boldsymbol{x}) = L'(f^d) \frac{\partial f^d(\boldsymbol{x})}{\partial \omega_{ij}^d}(\boldsymbol{x}) = L'(f^d(\boldsymbol{x})) o_i^d(\boldsymbol{x}) f_j^{d-1}(\boldsymbol{x}),$$

where $o^d = \sigma_d'(\boldsymbol{W}^d f^{d-1}(\boldsymbol{x}) + \boldsymbol{b}_d)$ and $L' : \boldsymbol{x} \to \frac{\partial L}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{y})$. It is then possible to obtain the derivatives of L with respect to parameters of the $d-1$-th layer, $\{\omega_{ij}^{d-1}\}$, $i, j \in \{1, ..., n_{d-1}\}$ by using the chain rule:

$$\frac{\partial L}{\partial \omega_{ij}^{d-1}}(\boldsymbol{x}) = L'(f^d(\boldsymbol{x})) \frac{\partial f^d}{\partial f^{d-1}}(\boldsymbol{x}) \frac{\partial f^{d-1}}{\partial \omega_{ij}^{d-1}}(\boldsymbol{x})$$

This formula is recursive, and for each layer, we have :

$$\begin{cases} \dfrac{\partial f^k}{\partial f^{k-1}}(\boldsymbol{x}) = o^k(\boldsymbol{x}) \boldsymbol{W}^k \\ \dfrac{\partial f^{k-1}}{\partial \omega_{ij}^{k-1}}(\boldsymbol{x}) = o_i^{k-1}(\boldsymbol{x}) f_j^{k-2}(\boldsymbol{x}). \end{cases}$$

Finally, $\nabla_{\boldsymbol{\theta}} L$ is computed backward by using the prediction $f^d(\boldsymbol{x})$, the weights of the network with $\frac{\partial f^k}{\partial f^{k-1}}(\boldsymbol{x})$, and the derivatives of the activation functions with $o_k(\boldsymbol{x})$. It is particularly efficient since this computation can be cast into matrix-vector products, just like the forward pass.

Most of the modern optimization algorithms used in deep learning are derived from stochastic gradient descent. We refer to 5.1 for a review of these algorithms.

2.4 Importance of the training data, hyperparameters, and optimization

In the previous parts, we have introduced the main building blocks of supervised deep learning: the formulation of supervised learning problems based on estimation from training data, the construction of the neural network's architecture, and its training using optimization algorithms. In the following, we emphasize the role of each block in deep learning and discuss their importance for supervised learning in scientific computing.

2.4.1 Importance for supervised learning

Let us briefly remind the task of supervised deep learning. The aim is to find $\boldsymbol{\theta}^*$ such that $J_{\mathbf{x}}(\boldsymbol{\theta}^*) \leq J_{\mathbf{x}}(\boldsymbol{\theta}) \ \ \forall \boldsymbol{\theta} \in \Theta$ by minimizing $\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta}) \approx J_{\mathbf{x}}(\boldsymbol{\theta})$. The specificity of supervised learning, and what makes this task so challenging, is that the function that has to be minimized ($J_{\mathbf{x}}$) is not the function on which optimization algorithms are applied ($\widehat{J}_{\mathbf{x}}$). Hence, the global minimum of $\widehat{J}_{\mathbf{x}}$, that we note $\hat{\boldsymbol{\theta}}^*$, may be different from $\boldsymbol{\theta}^*$. However, the optimization being non-convex, we have no guarantees that the value returned by actual optimizers is the global minimum. Instead, the optimizer returns a local minimum $\hat{\boldsymbol{\theta}}$. To sum up, we have :

- $\boldsymbol{\theta}^*$, the global minimum of $J_{\mathbf{x}}(\boldsymbol{\theta})$,
- $\hat{\boldsymbol{\theta}}^*$, the global minimum of $\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta})$,
- $\hat{\boldsymbol{\theta}}$, a local minimum of $\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta})$ found by the optimizer,

and the final error of the neural network is $J_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$. Now, we can decompose $J_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$ into four different parts, which can be identified according to their origin :

$$J_{\mathbf{x}}(\hat{\boldsymbol{\theta}}) = \underbrace{J_{\mathbf{x}}(\boldsymbol{\theta}^*)}_{Architecture} + \underbrace{\left(\widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}}) - \widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}}^*)\right)}_{Optimization} + \underbrace{\left(J_{\mathbf{x}}(\hat{\boldsymbol{\theta}}) - \widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}})\right)}_{Generalization} + \underbrace{\left(\widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}}^*) - J_{\mathbf{x}}(\boldsymbol{\theta}^*)\right)}_{Estimation}. \tag{2.6}$$

Architecture error. Universal approximation theorems (Hornik et al., 1989; Barron, 1994; Lu et al., 2017) state that when approximating a function f with a neural network $f_{\boldsymbol{\theta}}$, for any ϵ, there is a network whose width or depth allows approximating f such that $J_{\mathbf{x}}(\boldsymbol{\theta}^*) < \epsilon$.

However, in practice, we choose the width, the depth, and other hyperparameters beforehand. The error $J_{\mathbf{x}}(\boldsymbol{\theta}^*)$ is the minimum error that can be achieved for a given architecture, hence the name of architecture error. This error is also referred to as approximation error (Bottou and Bousquet, 2008).

Optimization error. As we discussed in the previous section, the minimization of $\widehat{J}_{\mathbf{x}}$ is a non-convex optimization problem, with many local minima. Therefore, the optimizer often returns a local minimum $\hat{\boldsymbol{\theta}}$, resulting in an additional error corresponding to the difference between $\widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$, the value of $\widehat{J}_{\mathbf{x}}$ at the local minimum, and $\widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}}^*)$, its actual minimum value.

Generalization error. At the end of the training, the optimizer returns $\hat{\boldsymbol{\theta}}$. Even if it is only a local, and not a global, minimum of $\widehat{J}_{\mathbf{x}}(\boldsymbol{\theta})$, the value obtained for the error $\widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$ is usually quite satisfying. However, the error obtained on the training data may be different from that obtained on unseen data. This difference, $J_{\mathbf{x}}(\hat{\boldsymbol{\theta}}) - \widehat{J}_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$, is sometimes called the generalization gap, or generalization error. This error may become so large that the obtained model is not usable. This problem is called overfitting and is well known in machine learning. Many techniques have been set up to limit this problem, such as cross-validation, weight decay (Krogh and Hertz, 1992), dropout (Srivastava et al., 2014), early stopping (Caruana et al., 2001)... But it will last as long as we do not have access to $J_{\mathbf{x}}(\hat{\boldsymbol{\theta}})$, that is to say forever.

Estimation error. In the hypothetic case where we would have access to a perfect optimizer capable of finding the global minimum of a non-convex function, there would still be an error because we do not explicitly optimize the function that we aim at minimizing ($J_{\mathbf{x}}$), but an estimation of this function ($\widehat{J}_{\mathbf{x}}$). The error that comes from inaccuracies in the estimation of $J_{\mathbf{x}}$ can be called estimation error. Note that this estimation error looks like the generalization error, in the sense that it comes from a difference between $J_{\mathbf{x}}$ and $\widehat{J}_{\mathbf{x}}$. However, the errors are not evaluated at the same value of $\boldsymbol{\theta}$. The estimation error and the generalization errors could be respectively high and low at the same time.

All these errors emphasize the importance of the steps described in the previous sections. The estimation and generalization errors stress the importance of the training data: we can hope to reduce these errors by working on the construction of the data set and on the estimation of $J_{\mathbf{x}}$. The architecture error points out the importance of hyperparameters, especially in deep learning, where they are so numerous. Choosing a good architecture allows reducing the architecture error. Finally, the optimization error illustrates the impact of the optimizer on the final error.

Remark. *The separation of the links between the error decomposition on one side and the training data, hyperparameters, and optimization on the other side is not clear-cut. For instance, some optimization algorithms aim at finding minima that generalize better (Izmailov et al., 2018b), some architectures are designed to ease the optimization (He et al., 2015), and there can be several hyperparameters in optimizers and data preprocessing techniques.*

Nonetheless, this error decomposition is still true, and it is helpful to emphasize the primary role of the training data, hyperparameters, and optimization in achieving good errors.

2.4.2 Importance for hybrid numerical simulations

By decomposing the loss function, we emphasized the role of the training data, hyperparameters, and optimization in deep learning. The objective of using supervised deep learning to accelerate numerical simulations requires analyzing each of these steps while considering the specificities and the constraints of this field.

One of the specificities of learning in numerical simulations is that the training data used to perform machine learning is simulated. Unlike typical machine learning problems where only a set of real-world data acquired experimentally is accessible, we can construct our database. Therefore, it is natural to think about methodologies for sampling training data when using deep learning in numerical simulations. Moreover, when seeking to accelerate computations using deep learning, the cost efficiency of neural networks becomes a concern because the execution time of a hybrid code directly depends on the execution time of neural networks used in that code. This concern reinforces the interest in working on the sampling of the training data. Indeed, it is a way of reducing the prediction error without any cost in the execution time.

Recent impressive performances of deep learning can be explained, among other reasons, by the ability to construct broader and deeper networks. However, equation (2.2) shows that the cost of one neural network's prediction increases quadratically with the width and linearly with the depth. These two hyperparameters are essential because they both have an impact on the accuracy and on the execution time of neural networks. More generally, there are many hyperparameters in deep learning, all impacting the error, and many of them impacting the execution time. It justifies carefully selecting them and thinking about selection methodologies.

Finally, as we shall see in Chapters 4 and 6, and as testified by the emulation in research on non-convex optimization, the optimizer is essential to reduce the prediction error. Moreover, reducing the error by improving the optimization has no impact on the final cost-efficiency of the neural network. These points are strong incentives to study the optimization of neural networks.

2.5 Discussion

In this chapter, we have introduced the notions needed to follow the approach of supervised deep learning. These notions are the formulation of the learning task using training data, the architecture defined by hyperparameters, and the training performed by the resolution of a non-convex optimization problem. They can be seen as steps for the methodology of

supervised deep learning. Indeed, one first needs to define the learning problem, then define an architecture for the neural network, and finally train this network.

Remark. *Although the three steps previously described have their own importance and specificities, the second one, hyperparameters selection, generally encompasses the two others. Indeed, there may be hyperparameters in the way the database is constructed as well as in the optimization process.*

Each step is crucial and carries its challenges and stakes. Moreover, intending to apply them in the context of numerical simulation brings about additional concerns due to the specificities and constraints of the field. All this justifies studying each step carefully before applying them to the construction of hybrid numerical simulation codes.

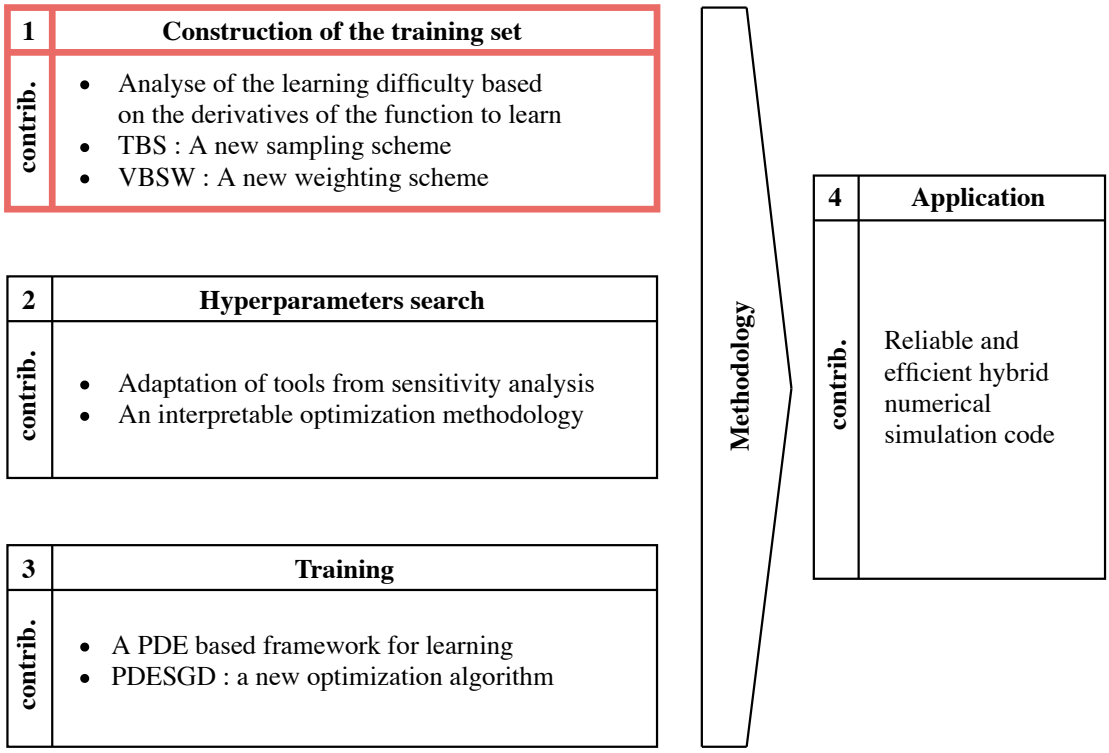
Chapter 3

Training distribution and local variation

In this chapter, we leverage the importance of the training set distribution to improve the performances of neural networks in supervised deep learning. Recall that the aim of supervised learning is to approximate a function f with a model $f_{\boldsymbol{\theta}}$ parametrized by $\boldsymbol{\theta}$ using data points drawn from $X \sim d\mathbb{P}_X$, $X \in \mathcal{X}$. We build a new distribution $d\mathbb{P}_{\bar{X}}$ from the training points and their labels, based on the observation that $f_{\boldsymbol{\theta}}$ needs more data points to approximate f on the regions where it is steep. We derive an illustrative generalization bound involving the derivatives of f that theoretically corroborates this observation. Therefore, we build $d\mathbb{P}_{\bar{X}}$ using Taylor expansion of the function f, which links the local behavior of f to its derivatives.

We first focus on the influence of using $d\mathbb{P}_{\bar{X}}$ instead of $d\mathbb{P}_X$ in simple approximation problems. To that end, we build a methodology for constructing and exploiting $d\mathbb{P}_{\bar{X}}$, that we call Taylor Based sampling (TBS). We then apply TBS to a more realistic problem based on the approximation of the solution of Bateman equations. Solving these equations is an important part of many numerical simulations of several phenomena (neutronic (Bernède and Poëtte, 2018; Dufek et al., 2013), combustion (Bisi and Desvillettes, 2006), detonic (Lucor et al., 2007a), computational biology (Perthame, 2007), etc.).

Then, we study the benefits of this approach for more general machine learning problems. In these cases, exploiting $d\mathbb{P}_{\bar{X}}$ is less straightforward. Indeed, we do not know the derivatives of f, and we cannot obtain labels for new data points sampled from this distribution. To tackle these problems, we show that variance is an approximation of Taylor expansion up to a certain order. Then we leverage the link between sampling and weighting to construct a methodology called Variance Based Sample Weighting (VBSW). We specifically investigate its application in deep learning, where we apply VBSW within the feature space of a pre-trained neural network. We validate VBSW by obtaining performance improvements on various tasks like classification and regression of text, from Glue benchmark (Wang et al., 2019), image, from MNIST (LeCun and Cortes, 2010) and Cifar10 (Krizhevsky et al.) and multivariate data, from UCI machine learning repository (Dua and Graff, 2017), for several models ranging from linear regression to Bert (Devlin et al., 2019) or ResNet20 (He et al., 2015).



Methodology for supervised deep learning in numerical simulations and contributions of the thesis

Contents

3.1 Related works

This work introduces contributions that rely on different elements. First, many techniques aim at altering the training distribution to improve the prediction error of neural networks. Second, finding generalization bounds for neural networks is the goal of various works in machine learning research. Finally, the methodology of constructing a sampling distribution for statistical analysis is used for importance sampling and designs of experiments.

Modified learning distributions Some works are dedicated to improving neural network performances by modifying the training distribution, either by weighting data points or by inducing sample selection. Active learning (Settles, 2012) adapts the training strategy to a learning problem by introducing an online data point selection rule. Gal et al. (2017) uses the variational properties of Bayesian neural network to design a rule that focuses the training on points that will reduce the prediction uncertainty of the neural network. In Konyushkova et al. (2017), the construction of the selection rule is itself taken as a machine learning problem. See Settles (2012) for a review of more classical active learning methods. Unlike active learning, and similarly to VBSW, some other methods aim at introducing diverse *a priori* evaluations of sample importance. While curriculum learning (Bengio et al., 2009a; Matiisen et al., 2017) starts the training with easier examples, Self-paced learning (Kumar et al., 2010; Jiang et al., 2015) downscales harder examples. However, some works have proven that focusing on harder examples at the beginning of the learning could accelerate it: Shrivastava et al. (2016) performs hard example mining to give more importance to harder examples by selecting them primarily. This work also focuses on defining hard examples but does so with an original, mathematical way based on f derivatives and local variance. It also stands out from the aforementioned techniques for how it modifies the distribution based on this information. Indeed, it suggests and justifies that a neural network should spend more learning time on subspaces of $\mathcal{X}$ which contain harder examples.

Generalization bounds As an argument to motivate our approach, we derive a generalization bound. The construction of Generalization bounds for the learning theory of neural networks has motivated many works (see Jakubovitz et al. (2018) for a review). In Bartlett et al. (1998, 2019), the authors focus on VC-dimension, a measure that depends on the number of parameters of neural networks. Arora et al. (2018) introduces a compression approach that

aims at reducing the number of model parameters to investigate its generalization capacities. PAC-Bayes analysis constructs generalization bounds using *a priori* and *a posteriori* distributions over the possible models. It is investigated, for example, in Neyshabur et al. (2018); Bartlett et al. (2017). Neyshabur et al. (2017); Xu and Mannor (2012) links PAC-Bayes theory to the notion of sharpness of a neural network, i.e. its robustness to small perturbation. While previous works often mention the sharpness of the model, our bound includes the derivatives of f, which can be seen as an indicator of the sharpness of the function to be learnt. Even if it uses elements of previous works, like the Lipschitz constant of $f_{\boldsymbol{\theta}}$, our work does not pretend to tighten and improve the already existing generalization bounds. It only emphasizes the intuition that the neural network would need more points to capture sharper functions. In a sense, it investigates the robustness to perturbations in the input space, not in the parameter space.

Examples weighting VBSW can be categorized as an examples weighting, or importance weighting algorithm. The idea of weighting the data set has already been explored in different ways and for various purposes. Examples weighting is used in Cui et al. (2019) to tackle the class imbalance problem by weighting rarer, so harder examples. On the contrary, in Liu and Tao (2016) it is used to solve the noisy label problem by focusing on cleaner, so easier examples. All these ideas show that depending on the application, examples weighting can be performed in an opposed manner. Some works aim at going beyond this opposition by proposing more general methodologies. In Chang et al. (2017), the authors use the variance of the prediction of each point throughout the training to decide whether it should be weighted or not. A meta-learning approach is proposed in Ren et al. (2018), where the authors choose the weights after an optimization loop included in the training. VBSW stands out from the previously mentioned examples weighting methods because it does not aim at solving dataset-specific problems like class imbalance or noisy labels. It is built on a more general assumption that a model would simply need more points to learn more complicated functions. The resulting weighting scheme verifies recent findings of Xu et al. (2021) where authors conclude that in classification, a good set of weights would put importance on points close to the decision boundary.

Importance sampling The challenge of finding a good distribution is not specific to machine learning. Indeed, in the context of Monte Carlo estimation of a quantity of interest based on a random variable, importance sampling owes its efficiency to the construction of a second random variable, which is used instead to improve the estimation of this quantity. Jie and Abbeel (2010) even make a connection between the success of likelihood ratio policy gradients and importance sampling, which shows that machine learning and Monte Carlo estimation, both distribution-based methods, are closely linked. Moreover, some previously mentioned methods use importance sampling to design the weights of the data set or to correct the bias induced by the sample selection (Katharopoulos and Fleuret, 2018). In this work, we construct a new distribution that could be interpreted as an importance distribution. However, we weigh the data points to simulate this distribution. It does not aim at correcting a bias induced by this distribution.

Designs of experiments Some methodologies are dedicated to the construction of data sets in the context of statistical analysis. These methodologies are called designs of experiments. In our case, the construction of a new training distribution could be seen as a design of experiments for learning. However, popular designs of experiments used for regression are either space-filling designs or model-based designs. Space-filling designs, like latin hypercube sampling (McKay et al., 1979) or maximin designs (Johnson et al., 1990), aims at spreading the learning points to cover the input space as much as possible. Model-based designs use characteristics of $f_{\boldsymbol{\theta}}$ to adapt the training distribution. Such designs can look to maximize the entropy of the prediction (Shewry and Wynn, 1987) or minimize its uncertainty (Jones et al., 1998b). These last designs of experiments can be conducted sequentially, getting close to active learning (Seo et al., 2000; MacKay, 1992; Cohn, 1996). Our methodology does not depend on $f_{\boldsymbol{\theta}}$, nor aims at filling the input space. Instead, its goal is to adapt the design of experiments to characteristics of f in order to reduce the prediction error.

3.2 Link between local variations and learning

Let us first remind some basics on supervised machine learning. We formalize the supervised machine learning task as approximating a function $f : \mathbf{S} \subset \mathbb{R}^{n_i} \to \mathbb{R}^{n_o}$ with a machine learning model $f_{\boldsymbol{\theta}}$ parametrized by $\boldsymbol{\theta}$, where $\mathbf{S}$ is a measured sub-space of $\mathbb{R}^{n_i}$ depending on the application. To this end, we are given a training data set of N points, $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\} \in \mathbf{S}$, drawn from $\mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}$ and their point-wise values, or labels $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$. Parameters $\boldsymbol{\theta}$ have to be found in order to minimize an integrated loss function $J_{\mathbf{x}}(\boldsymbol{\theta}) = \mathbb{E}[L(f_{\boldsymbol{\theta}}(\boldsymbol{x}), f(\boldsymbol{x}))]$, with L the loss function, $L : \mathbb{R}^{n_o} \times \mathbb{R}^{n_o} \to \mathbb{R}$. The data allow estimating $J_{\mathbf{x}}(\boldsymbol{\theta})$ by $\widehat{J_{\mathbf{x}}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \omega_i L(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), f(\boldsymbol{x}_i))$, with $\{\omega_1, ..., \omega_N\} \in \mathbb{R}$ estimation weights, generally equal to $\frac{1}{N}$. Then, an optimization algorithm is used to find a minimum of $\widehat{J_{\mathbf{x}}}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$.

3.2.1 Illustration of the link using derivatives

In the following, we illustrate the intuition with a Generalization Bound (GB) that include the derivatives of f, provided that these derivatives exist. The goal of the approximation problem is to be able to generalize to points not seen during the training. The generalization error $\mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta}) = J_{\mathbf{x}}(\boldsymbol{\theta}) - \widehat{J_{\mathbf{x}}}(\boldsymbol{\theta})$ thus needs to be as small as possible. Let S_i, $i \in \{1, ..., N\}$ be some sub-spaces of $\mathbf{S}$ such that $\mathbf{S} = \bigcup_{i=1}^{N} S_i$, $\bigcap_{i=1}^{N} S_i = \emptyset$, and $\boldsymbol{x}_i \in S_i$. Suppose that L is the squared L_2 error, $n_i = n_o = 1$, f is differentiable, $f_{\boldsymbol{\theta}}$ is $K_{\boldsymbol{\theta}}$-Lipschitz and satisfies the conditions of Hornik theorem Hornik et al. (1989). Provided that $|S_i| < 1$, we show that

$$\mathcal{J}_{\mathrm{x}}(\boldsymbol{\theta}) \leq \sum_{i=1}^{N} (|f'(x_i)| + K_{\boldsymbol{\theta}})^2 \frac{|S_i|^3}{4} + \mathcal{O}(|S_i|^4), \tag{3.1}$$

where $|S_i|$ is the volume of S_i ($|S_i| = \int_{S_i} d\mathbb{P}_{\mathbf{x}}$). The proof can be found in **Appendix A**. We see that in the regions where $f'(\boldsymbol{x}_i)$ is high, quantity $|S_i|$ has a stronger impact on the GB. This idea is illustrated in Figure 3.1, which visually shows that the generalization bound increases when $|S_i|$ and $f'(\boldsymbol{x}_i)$ are high at the same time for approximating the function $f : x \to x^3$. Since $|S_i|$ can be seen as a metric for the local density of the data set (the smaller $|S_i|$ is, the denser the data set is), the GB can be reduced more efficiently by adding more points around $\boldsymbol{x}_i$ in these regions. This bound also involves $K_{\boldsymbol{\theta}}$, the Lipschitz constant of the neural network, which has the same impact as $f'(\boldsymbol{x}_i)$. It also illustrates the link between the Lipschitz constant and the generalization error, which has been pointed out by several works like Gouk et al. (2018), Bartlett et al. (2017) and Qian and Wegman (2019).

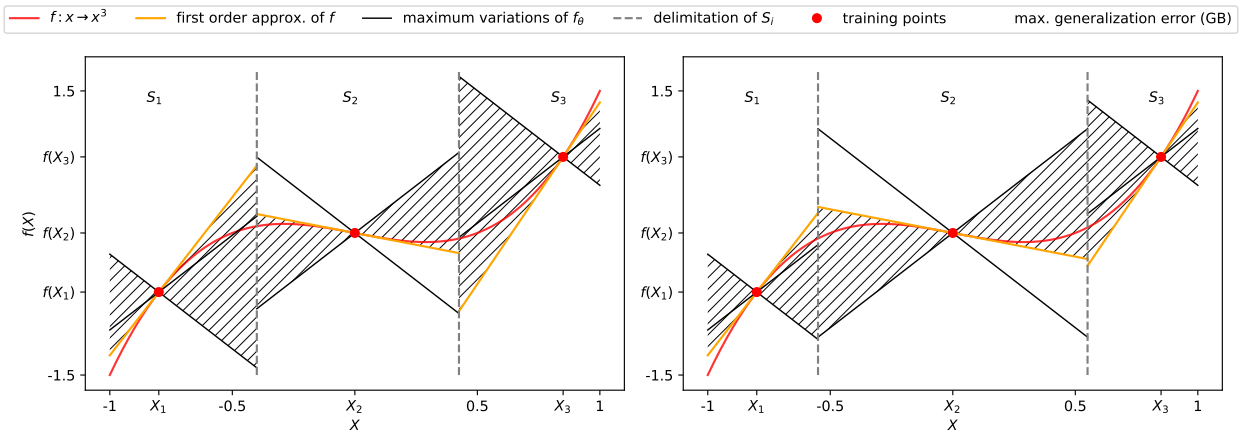


Figure 3.1: Illustration of the GB. The maximum error (the GB), at order $\mathcal{O}(|S_i|^4)$, is obtained by comparing the maximum variations of $f_{\boldsymbol{\theta}}$, and the first order approximation of f, whose trends are given by $K_{\boldsymbol{\theta}}$ and $f'(\boldsymbol{x}_i)$. We understand visually that because $f'(\boldsymbol{x}_1)$ and $f'(\boldsymbol{x}_3)$ are higher than $f'(\boldsymbol{x}_2)$, the GB is improved more efficiently by reducing S_1 and S_3 than S_2.

3.2.2 A sampling scheme based on Taylor Approximation

Equation (3.1) formalizes a link between generalization error and derivatives of f. These derivatives are expressed at order $n = 1$ for analytical reasons, but in this work we explore the use of derivatives of order $n > 1$. Using Taylor expansion at order n on f and supposing that f is n times differentiable:

$$f(\boldsymbol{x} + \boldsymbol{\epsilon}) \underset{\|\boldsymbol{\epsilon}\| \to 0}{=} \sum_{0 \leq |\boldsymbol{k}| \leq n} \boldsymbol{\epsilon}^{\boldsymbol{k}} \frac{\partial^{\boldsymbol{k}} f(\boldsymbol{x})}{\boldsymbol{k}!} + \mathcal{O}(\boldsymbol{\epsilon}^{\boldsymbol{n}}).$$

The quantity $f(\boldsymbol{x} + \boldsymbol{\epsilon}) - f(\boldsymbol{x}) = \sum_{1 \leq |\boldsymbol{k}| \leq n} \boldsymbol{\epsilon}^{\boldsymbol{k}} \frac{\partial^{\boldsymbol{k}} f(\boldsymbol{x})}{\boldsymbol{k}!} + \mathcal{O}(\boldsymbol{\epsilon}^{\boldsymbol{n}})$ gives an indication on how much f changes around $\boldsymbol{x}$. By neglecting the orders above $\boldsymbol{\epsilon}^{\boldsymbol{n}}$, it is then possible to find the regions of interest by focusing on $Df^n_{\boldsymbol{\epsilon}}$, defined as:

$$Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}) = \sum_{1\leq|\boldsymbol{k}|\leq n} \boldsymbol{\epsilon}^k \frac{(\partial^{\boldsymbol{k}} f(\boldsymbol{x}))^2}{\boldsymbol{k}!}, \tag{3.2}$$

where $\boldsymbol{k}$ is a multi-index, i.e. $\boldsymbol{k} = (k_1, ..., k_{n_i})$ is a vector of n_i non negative integers with $|\boldsymbol{k}| = \sum_{i=1}^{n_i} k_i$, $\boldsymbol{k}! = k_1! \times ... \times k_{n_i}!$, $\boldsymbol{\epsilon}^{\boldsymbol{k}} = \epsilon_1^{k_1} \times ... \times \epsilon_{n_i}^{k_{n_i}}$, $\partial^{\boldsymbol{k}} = \frac{\partial^{k_1}}{\partial x_1^{k_1}} \times ... \times \frac{\partial^{k_{n_i}}}{\partial x_{n_i}^{k_{n_i}}}$. Note that $Df^n_{\boldsymbol{\epsilon}}$ is evaluated using $(\partial^{\boldsymbol{k}} f(\boldsymbol{x}))^2$ instead of $\partial^{\boldsymbol{k}} f(\boldsymbol{x})$ for derivatives not to cancel each other. To avoid these cancellations, the absolute could have been used, but we will see in Lemma 1 that the square value ensures interesting asymptotical properties.c f will be steeper and more irregular in the regions where $\boldsymbol{x} \to Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x})$ is higher. To focus the training set on these regions, one can use $\{Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ to construct a probability density function (pdf) and sample new data points from it.

In this part, we empirically verify that using Taylor expansion to construct a new training distribution has a beneficial impact on the performances of a neural network. To this end, we construct a methodology, that we call Taylor Based Sampling (TBS), that generates a new training data set based on the metric equation (3.2). To focus the training set on the regions of interest, i.e. regions of high $\{Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$, we use this metric to construct a probability density function (pdf) - which is possible since $Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x} \in \mathbf{S}$. It remains to normalize it but in practice it is enough considering a distribution $d\mathbb{P}_{\bar{\mathbf{x}}} \propto Df^n_{\boldsymbol{\epsilon}}$. Here, to approximate $d\mathbb{P}_{\bar{\mathbf{x}}}$ we use a Gaussian Mixture Model (GMM) with pdf $d\mathbb{P}_{\bar{\mathbf{x}},GMM}$ that we fit to $\{Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ using the Expectation-Maximization (EM) algorithm. N' new data points $\{\bar{\boldsymbol{x}}_1, ..., \bar{\boldsymbol{x}}_{N'}\}$, can be sampled, with $\bar{\mathbf{x}} \sim d\mathbb{P}_{\bar{\mathbf{x}},GMM}$. Finally, we obtain $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$, add it to $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$ and train our neural network on the whole data set.

TBS is described in Algorithm 1. **Line 1:** The parameter $\boldsymbol{\epsilon}$, the number of Gaussian distribution n_{GMM} and N' is chosen in order to avoid sparsity of $\{\bar{\boldsymbol{x}}_1, ..., \bar{\boldsymbol{x}}_{N'}\}$ over $\mathbf{S}$. **Line 2:** Without *a priori* information on f, we sample the first points uniformly in a subspace $\mathbf{S}$. **Line 3-7:** We construct $\{Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$, and then $d\mathbb{P}_{\bar{\mathbf{x}},GMM}$ to be able to sample points accordingly. **Line 8:** Because the support of a GMM is not bounded, some points can be sampled outside $\mathbf{S}$. We discard these points and sample until all points are inside $\mathbf{S}$. This rejection method is equivalent to sampling points from a truncated GMM. **Line 9-10:** We construct the labels and add the new points to the initial data set.

3.2.3 Application to simple functions

To illustrate the benefits of TBS compared to a uniform, basic sampling (BS), we apply it to two simple functions: hyperbolic tangent and Runge function. We chose these functions because they are differentiable and have a clear distinction between flat and steep regions. These functions are displayed in Figure 3.2, as well as the map $\boldsymbol{x} \to Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x})$.

All neural networks have been implemented in `Python`, with `Tensorflow` Abadi et al. (2015).

Algorithm 1 Taylor Based Sampling (TBS)

Inputs: $\boldsymbol{\epsilon}$, N, N', n_{GMM}, n
Sample $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ from $\mathbf{x} \sim \mathcal{U}(\mathbf{S})$
for $0 \leq k \leq n$ **do**
    Compute $\{\partial^{\boldsymbol{k}} f(\boldsymbol{x}_1), ..., \partial^{\boldsymbol{k}} f(\boldsymbol{x}_N)\}$
Compute $\{Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^n_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ using equation (3.2)
Approximate $d\mathbb{P}_{\bar{\mathbf{x}}} \propto Df^n_{\boldsymbol{\epsilon}}$ with a GMM using EM algorithm to obtain a density $d\mathbb{P}_{\bar{\mathbf{x}},GMM}$
Sample $\{\bar{\boldsymbol{x}}_1, ..., \bar{\boldsymbol{x}}_{N'}\}$ using rejection method to sample inside $\mathbf{S}$
Compute $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$
Add $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$ to $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$

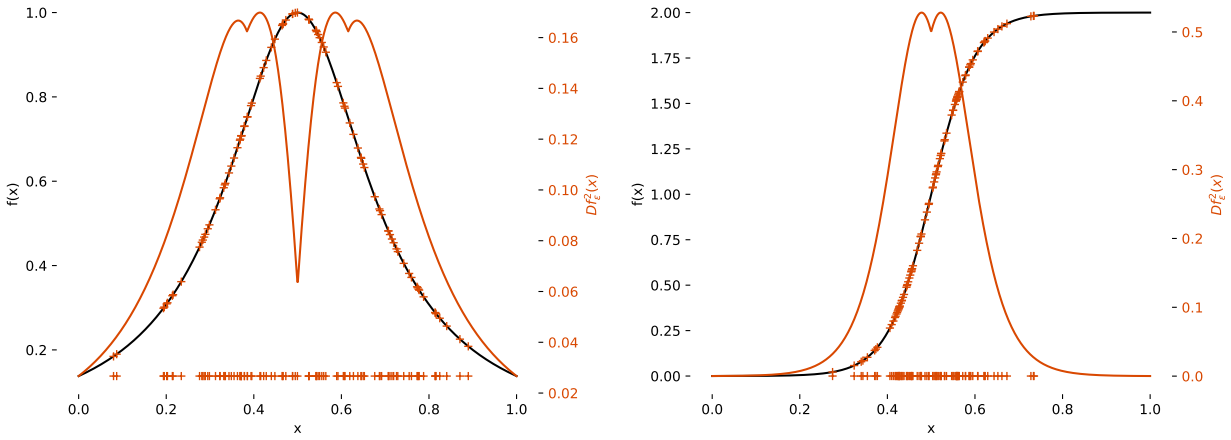


Figure 3.2: **Left:** (left axis) Runge function w.r.t x and (right axis) $x \to Df^2_{\boldsymbol{\epsilon}}(x)$. Points sampled using TBS are plotted on the x-axis and projected on f. **Right:** Same as left, with hyperbolic tangent function.

We use the Python package `scikit-learn` Pedregosa et al. (2011), and more specifically the `GaussianMixture` class to construct $d\mathbb{P}_{\bar{\mathbf{x}},GMM}$. The network chosen for this experiment is a Multi Layer Perceptron (MLP) with one layer of 8 neurons and relu activation function, that we trained alternatively with BS and TBS using Adam optimizer Kingma and Ba (2015) with the defaults tensorflow implementation hyperparameters, and Mean Squared Error loss function. We first sample $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ according to a regular grid. To compare the two methods, we add N' additional points sampled using BS to create the BS data set, and then N' other points sampled with TBS to construct the TBS data set. As a result, each data set have the same number of points $(N + N')$. We repeated the method for several values of n, n_{GMM} and $\boldsymbol{\epsilon}$, and finally selected $n = 2$, $n_{\text{GMM}} = 3$ and $\boldsymbol{\epsilon} = 10^{-3}$.

Table 3.1 summarizes the L_2 and the L_∞ norm of the error of $f_{\boldsymbol{\theta}}$, obtained at the end of the training phase for $N + N' = 16$, with $N = 8$. Those norms are estimated using the same test data set of 1000 points. The values are the means of the 40 independent experiments displayed with a 95% confidence interval. These results illustrate the benefits of TBS over BS. Table 3.1 shows that TBS does not significantly improve L_2 error, but does so for L_∞ error, which may explain the good results of VBSW for classification that we describe in

Sampling	L_2 error	L_∞ error
	f: Runge ($\times 10^{-2}$)	
BS	1.45 ± 0.62	5.31 ± 0.86
TBS	$\mathbf{1.13} \pm 0.73$	$\mathbf{3.87} \pm 0.48$
	f: tanh ($\times 10^{-1}$)	
BS	1.39 ± 0.67	2.75 ± 0.78
TBS	$\mathbf{0.95} \pm 0.50$	$\mathbf{2.25} \pm 0.61$

Table 3.1: Comparison between BS and TBS. The metrics used are the L_2 and L_∞ errors, displayed with a 95% confidence interval.

Section 3.5. Indeed, the accuracy will not be very sensitive to small output variations for a classification task since the output is rounded to 0 or 1. However, a high error increases the risk of misclassification, which can be limited by the reduction of L_∞.

3.2.4 Application to an ODE system

We apply TBS to a more realistic case: the approximation of the resolution of the Bateman equations, which is an ODE system :

$$\begin{cases} \partial_t u(t) &= v\boldsymbol{\sigma_a} \cdot \boldsymbol{\eta}(t) u(t), \\ \partial_t \boldsymbol{\eta}(t) &= v\boldsymbol{\Sigma_r} \cdot \boldsymbol{\eta}(t) u(t), \end{cases} \text{, with initial conditions } \begin{cases} u(0) = u_0, \\ \boldsymbol{\eta}(0) = \boldsymbol{\eta_0}. \end{cases},$$

with $u \in \mathbb{R}^+, \boldsymbol{\eta} \in (\mathbb{R}^+)^M, \boldsymbol{\sigma}_a^T \in \mathbb{R}^M, \boldsymbol{\Sigma}_r \in \mathbb{R}^{M \times M}$. Here, $f : (u_0, \boldsymbol{\eta_0}, t) \to (u(t), \boldsymbol{\eta}(t))$. For physical applications, M ranges from tens to thousands, but we consider the particular case $M = 1$ so that $f : \mathbb{R}^3 \to \mathbb{R}^2$, with $f(u_0, \eta_0, t) = (u(t), \eta(t))$, and $\sigma_a = \sigma_r = -0.45$. The advantage of $M = 1$ is that we have access to an analytic, cheap to compute solution for f. Of course, this particular case can also be solved using a classical ODE solver, which allows us to test it end to end. It can thus be generalized to higher dimensions ($M > 1$).

All neural network training instances have been performed in `Python`, with `Tensorflow`. We used a fully connected neural network with hyperparameters chosen using a simple grid search. The final values are: 2 hidden layers, relu activation function, and 32 units for each layer, trained with the Mean Squared Error (MSE) loss function using Adam optimization algorithm with a batch size of 50000, for 40000 epochs and on $N + N' = 50000$ points, with $N = N'$. We first trained the model for $(u(t), \eta(t)) \in \mathbb{R}$, with an uniform sampling (BS) ($N' = 0$), and then with TBS for several values of n, n_{GMM} and $\boldsymbol{\epsilon} = \epsilon(1, 1, 1)$, to be able to find good values. We finally select $\epsilon = 5 \times 10^{-4}$, $n = 2$ and $n_{\text{GMM}} = 10$. The data points used in this case have been sampled with an explicit Euler scheme. This experiment has been repeated 50 times to

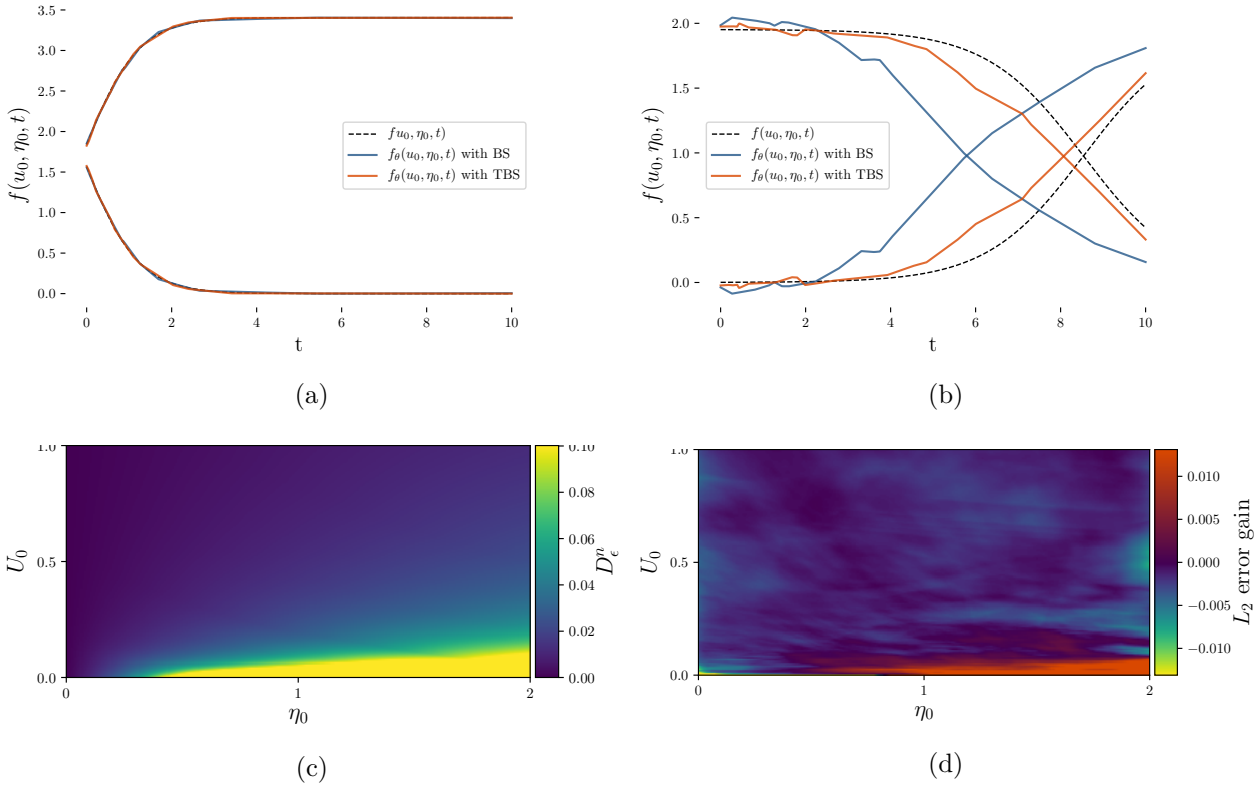


Figure 3.3: **(a)** $t \to f_{\boldsymbol{\theta}}(u_0, \eta_0, t)$ for randomly chosen (u_0, η_0), for $f_{\boldsymbol{\theta}}$ obtained with the two samplings. **(b)** $t \to f_{\boldsymbol{\theta}}(u_0, \eta_0, t)$ for (u_0, η_0) resulting in the highest point-wise error with the two samplings. **(c)** $u_0, \eta_0 \to \max_{0 \leq t \leq 10} D^n_{\boldsymbol{\epsilon}}(u_0, \eta_0, t)$ w.r.t. (u_0, η_0). **(d)** $u_0, \eta_0 \to g_{\theta_{BS}}(u_0, \eta_0) - g_{\theta_{TBS}}(u_0, \eta_0)$,

ensure statistical significance of the results.

Table 3.2 summarizes the MSE, i.e. the L_2 norm of the error of $f_{\boldsymbol{\theta}}$ and L_∞ norm, with $L_\infty(\boldsymbol{\theta}) = \max_{\boldsymbol{x} \in \mathbf{S}}(|f(\boldsymbol{x}) - f_{\boldsymbol{\theta}}(\boldsymbol{x})|)$ obtained at the end of the training phase. This last metric is important because the goal in computational physics is not only to be averagely accurate, which is measured with MSE, but to be accurate over the whole input space $\mathbf{S}$. Those norms are estimated using a same test data set of $N_{test} = 50000$ points. The values are the means of the 50 independent experiments displayed with a 95% confidence interval. These results reflect an error reduction of 6.6% for L_2 and of 45.3% for L_∞, which means that TBS mostly improves the L_∞ error of $f_{\boldsymbol{\theta}}$. Moreover, the L_∞ error confidence intervals do not intersect so the gain is statistically significant for this norm.

Table 3.2: Comparison between BS and TBS

Sampling	L_2 error ($\times 10^{-4}$)	L_∞ ($\times 10^{-1}$)	AEG($\times 10^{-2}$)	AEL($\times 10^{-2}$)
BS	1.22 ± 0.13	5.28 ± 0.47	-	-
TBS	$\mathbf{1.14} \pm 0.15$	$\mathbf{2.96} \pm 0.37$	2.51 ± 0.07	0.42 ± 0.008

Figure 3.3a shows how the neural network can perform for an average prediction. Figure

3.3b illustrates the benefits of TBS relative to BS on the L_∞ error (Figure 2b). These 2 figures confirm the previous observation about the gain in L_∞ error. Finally, Figure 3.3c displays $u_0, \eta_0 \rightarrow \max_{0 \leq t \leq 10} D^n_{\boldsymbol{\epsilon}}(u_0, \eta_0, t)$ w.r.t. (u_0, η_0) and shows that $D^n_{\boldsymbol{\epsilon}}$ increases when $U_0 \rightarrow 0$. TBS hence focuses on this region. Note that for the readability of these plots, the values are capped to 0.10. Otherwise only few points with high $D^n_{\boldsymbol{\epsilon}}$ are visible. Figure 3.3d displays $u_0, \eta_0 \rightarrow g_{\theta_{BS}}(u_0, \eta_0) - g_{\theta_{TBS}}(u_0, \eta_0)$, with $g_{\boldsymbol{\theta}} : u_0, \eta_0 \rightarrow \max_{0 \leq t \leq 10} \|f(u_0, \eta_0, t) - f_{\boldsymbol{\theta}}(u_0, \eta_0, t)\|_2^2$ where θ_{BS} and θ_{TBS} denote the parameters obtained after a training with BS and TBS, respectively. It can be interpreted as the error reduction achieved with TBS.

The highest error reduction occurs in the expected region. Indeed, more points are sampled where $D^n_{\boldsymbol{\epsilon}}$ is higher. The error is slightly increased in the rest of $\mathbf{S}$, which could be explained by a sparser sampling on this region. However, as summarized in **Table 1**, the average error loss (AEL) of TBS is around six times lower than the average error gain (AEG), with $AEG = \mathbb{E}[Z(u_0, \eta_0)\mathbf{1}_{Z>0}]$ and $AEL = \mathbb{E}[Z(u_0, \eta_0)\mathbf{1}_{Z<0}]$ where $Z(u_0, \eta_0) = g_{\theta_{BS}}(u_0, \eta_0) - g_{\theta_{TBS}}(u_0, \eta_0)$. In practice, AEG and AEL are estimated using uniform grid integration, and averaged on the 50 experiments.

3.3 Generalization of Taylor based Sampling

The previous section empirically validated the intuition behind the construction of a new, more efficient training distribution $d\mathbb{P}_{\tilde{\mathbf{x}}}$. However, this new distribution cannot always be applied as-is for two reasons. **Problem 1:** $\{Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ cannot be evaluated since it requires to compute the derivatives of f, and it assumes that f is differentiable, which is often not true. Moreover, the previously described setting, in which we focus on f derivatives, is not suited to classification tasks where the notion of derivatives is not straightforward. **Problem 2:** even if $\{Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ could be computed and new points sampled, we could not obtain their labels to complete the training data set. In this section, we alleviate this concern to be able to use insights from $d\mathbb{P}_{\tilde{\mathbf{x}}}$ in practice.

3.3.1 From Taylor expansion to local variance

To overcome **problem 1**, we construct a new metric based on statistical estimation. In this paragraph, $n_i > 1$ but $n_o = 1$. The following derivations can be extended to $n_o > 1$ by applying it to f element-wise and then taking the sum across the n_o dimensions. Let $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \epsilon \boldsymbol{I}_{n_i})$ with $\epsilon \in \mathbf{R}^+$ and $\boldsymbol{I}_{n_i}$ the identity matrix of dimension n_i. We claim that

Lemma 1. *Let* $\mathbf{e} \sim \mathcal{N}(0, \epsilon \boldsymbol{I}_{n_i})$ *with* $\epsilon \in \mathbf{R}^+$ *and* $\boldsymbol{I}_{n_i}$ *the identity matrix of dimension* n_i*. Let* $\boldsymbol{\epsilon} = \epsilon(1, ..., 1)$*. Then,*

$$Var(f(\boldsymbol{x} + \mathbf{e})) = Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3).$$

The demonstration can be found in **Appendix A**. Using the unbiased estimator of variance, we thus define new indices $\widehat{Df^2_{\boldsymbol{\epsilon}}}(\boldsymbol{x})$ by

$$\widehat{Df^2_{\boldsymbol{\epsilon}}}(\boldsymbol{x}) = \frac{1}{k-1}\sum_{j=1}^{k}\Big(f(\boldsymbol{x}+\boldsymbol{\epsilon_j}) - \frac{1}{k-1}\sum_{l=1}^{k} f(\boldsymbol{x}+\boldsymbol{\epsilon_l})\Big)^2, \tag{3.3}$$

with $\{\boldsymbol{\epsilon}_1, ..., \boldsymbol{\epsilon}_k\}$ k samples of $\boldsymbol{\epsilon}$. The metric $\widehat{Df^2}_{\epsilon}(\boldsymbol{x}) \underset{k\to\infty}{\to} Var(f(\boldsymbol{x}+\boldsymbol{\epsilon}))$ and $Var(f(\boldsymbol{x}+\boldsymbol{\epsilon})) = Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3)$, so $\widehat{Df^2}_{\epsilon}(\boldsymbol{x})$ is a biased estimator of $Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x})$, with bias $\mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3)$. Hence, when $\boldsymbol{\epsilon} \to 0$, $\widehat{Df^2}_{\epsilon}(\boldsymbol{x})$ becomes an unbiased estimator of $Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x})$. It is possible to compute $\widehat{Df^2}(\boldsymbol{x})$ from any set of points centered around $\boldsymbol{x}$. Therefore, we compute $\widehat{Df^2}(\boldsymbol{x}_i)$ for each $i \in \{1, ..., N\}$ using the set $\mathcal{S}_k(\boldsymbol{x})$ of k-nearest neighbors of $\boldsymbol{x}$. We obtain $\widehat{Df^2}(\boldsymbol{x}_i)$ using

$$\widehat{Df^2}(\boldsymbol{x}_i) = \frac{1}{k-1}\sum_{\boldsymbol{x}_j \in \mathcal{S}_k(\boldsymbol{x}_i)}\Big(f(\boldsymbol{x}_j) - \frac{1}{k}\sum_{\boldsymbol{x}_l \in \mathcal{S}_k(\boldsymbol{x}_i)} f(\boldsymbol{x}_l)\Big)^2, \tag{3.4}$$

which is equation (3.3) where we replaced $f(\boldsymbol{x}_i + \boldsymbol{\epsilon_j})$ with $f(\boldsymbol{x}_j)$ - and $f(\boldsymbol{x}_i + \boldsymbol{\epsilon_l})$ with $f(\boldsymbol{x}_l)$ - where $\boldsymbol{x}_j, \boldsymbol{x}_l \in \mathcal{S}_k(\boldsymbol{x}_i)$ are the neighbors of $\boldsymbol{x}_i$. Equation (3.4) has several practical advantages. First, $\widehat{Df^2}$ can even be applied to non-differentiable functions and for classification problems, unlike equation (3.2). Second, the definition of $\widehat{Df^2}(\boldsymbol{x})$ does not rely on $\boldsymbol{\epsilon}$, unlike equation (3.3). To compute $\widehat{Df^2}$, all we need are $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$, the points used for the training of the neural network. In addition, equation (3.4) can even be applied when the data points are too sparse for the nearest neighbors of $\boldsymbol{x}_i$ to be considered as close to $\boldsymbol{x}_i$, which is almost always the case in high dimension. It can thus be seen as a generalization of $\widehat{Df^2_{\boldsymbol{\epsilon}}}(\boldsymbol{x})$, which tends towards $Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x})$ locally.

3.3.2 From sampling to weighting

To tackle **problem 2**, recall that the goal of the training is to find $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \widehat{J_{\mathbf{x}}}(\boldsymbol{\theta})$, with $\widehat{J_{\mathbf{x}}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_i L(f(\boldsymbol{x}_i), f_{\boldsymbol{\theta}}(\boldsymbol{x}_i))$. With the new distribution based on previous derivations, the procedure is different. Since the training points are sampled using $\widehat{Df^2}_{\boldsymbol{\epsilon}}$, we no longer minimize $\widehat{J_{\mathbf{x}}}(\boldsymbol{\theta})$, but $\widehat{J_{\bar{\mathbf{x}}}}(\boldsymbol{\theta}) = \frac{1}{N}\sum_i L(f(\bar{\boldsymbol{x}}_i), f_{\boldsymbol{\theta}}(\bar{\boldsymbol{x}}_i))$, with $\bar{\mathbf{x}} \sim d\mathbb{P}_{\bar{\mathbf{x}}}$ the new distribution. However, $\widehat{J_{\bar{\mathbf{x}}}}(\boldsymbol{\theta})$ estimates

$$J_{\bar{\mathbf{x}}}(\boldsymbol{\theta}) = \int_{\mathbf{S}} L(f(\boldsymbol{x}), f_{\boldsymbol{\theta}}(\boldsymbol{x}))d\mathbb{P}_{\bar{\mathbf{x}}}.$$

Let $p_{\mathbf{x}}(\boldsymbol{x})d\boldsymbol{x} = d\mathbb{P}_{\mathbf{x}}$, $p_{\bar{\mathbf{x}}}(\boldsymbol{x})d\boldsymbol{x} = d\mathbb{P}_{\bar{\mathbf{x}}}$ be the pdfs of $\mathbf{x}$ and $\bar{\mathbf{x}}$ (note that $Df_{\boldsymbol{\epsilon}}^2 \propto p_{\bar{\mathbf{x}}}$). Then,

$$J_{\bar{\mathbf{x}}}(\boldsymbol{\theta}) = \int_{\mathbf{S}} L(f(\boldsymbol{x}), f_{\boldsymbol{\theta}}(\boldsymbol{x})) \frac{p_{\bar{\mathbf{x}}}(\boldsymbol{x})}{p_{\mathbf{x}}(\boldsymbol{x})} d\mathbb{P}_{\mathbf{x}}.$$

The straightforward Monte Carlo estimator for this expression of $J_{\bar{\mathbf{x}}}(\boldsymbol{\theta})$ is

$$\widehat{J_{\bar{\mathbf{x}}}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_i L(f(\boldsymbol{x}_i), f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)) \frac{p_{\bar{\mathbf{x}}}(\boldsymbol{x}_i)}{p_{\mathbf{x}}(\boldsymbol{x}_i)} \propto \frac{1}{N} \sum_i L(f(\boldsymbol{x}_i), f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)) \frac{\widehat{Df^2}(\boldsymbol{x}_i)}{p_{\mathbf{x}}(\boldsymbol{x}_i)}. \tag{3.5}$$

Thus, $J_{\bar{\mathbf{x}}}(\boldsymbol{\theta})$ can be estimated with the same points as $J_{\mathbf{x}}(\boldsymbol{\theta})$ by weighting them with $w_i = \frac{\widehat{Df^2}(\boldsymbol{x}_i)}{p_{\mathbf{x}}(\boldsymbol{x}_i)}$.

The expression of w_i involves $p_{\mathbf{x}}$, the distribution of the data. Just like for f, we do not have access to $p_{\mathbf{x}}$. The estimation of $p_{\mathbf{x}}$ is a challenging task by itself, and standard density estimation techniques such as K-nearest neighbors or Gaussian Mixture density estimation led to extreme estimated values of $p_{\mathbf{x}}(\boldsymbol{x}_i)$ in our experiments. Therefore, we decided to only apply $\omega_i = \widehat{Df^2}(\boldsymbol{x}_i)$ as a first-order approximation. In practice, we re-scale the weights between 1 and m, a hyperparameter, and then divide them by their sum to avoid affecting the learning rate.

As a result, we obtain a new methodology based on weighting the training data set. We call this methodology Variance Based Sample Weighting (VBSW).

3.4 Variance Based Sample Weighting

In this part, we sum up Variance Based Sample Weighting (VBSW) to clarify its application to machine learning problems. We also study this methodology through toy experiments.

3.4.1 Methodology

Variance Based Samples Weighting (VBSW) is recapitulated in Algorithm 2. **Line 1:** m and k are hyperparameters that can be chosen jointly with all other hyperparameters, e.g. using a random search. Their effects and interactions are studied and discussed in Sections 3.4.2 and 3.5.4. **Line 2-3:** equation (3.4) is applied to compute the weights w_i that are used to weight the data set. Notations $\{(w_1, \boldsymbol{x}_1), ..., (w_N, \boldsymbol{x}_N)\}$ denote that each $\boldsymbol{x}_i$ is weighted by w_i. To perform a nearest-neighbors search, we use an approximate nearest neighbor search technique called hierarchical navigable small world graphs (Malkov and Yashunin, 2020) implemented by nmslib (Boytsov and Naidan, 2013). **Line 4:** Train $f_{\boldsymbol{\theta}}$ on the weighted data set.

Algorithm 2 Variance Based Samples Weighting (VBSW)

1: **Inputs:** k, m
2: Compute $\{\widehat{Df^2}(\boldsymbol{x}_1), ..., \widehat{Df^2}(\boldsymbol{x}_N)\}$ using equation (3.4).
3: Construct a new training data set $\{(w_1, \boldsymbol{x}_1), ..., (w_N, \boldsymbol{x}_N)\}$
4: Train $f_{\boldsymbol{\theta}}$ on $\{(w_1, f(\boldsymbol{x}_1)), ..., (w_N, f(\boldsymbol{x}_N))\}$

3.4.2 Toy experiments & hyperparameter study

VBSW is studied on a Double Moon (DM) classification problem, the Boston Housing (BH) regression, and Breast Cancer (BC) classification data sets.

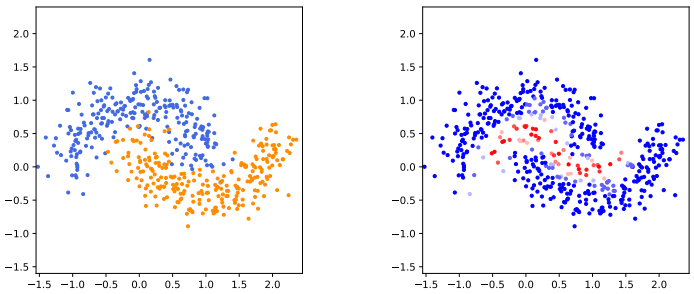


Figure 3.4: From left to right: **(a)** Double Moon (DM) data set. **(b)** Heat map of the value of w_i for each $\boldsymbol{x}_i$ (red is high and blue is low)

For DM, Figure 3.4 (c) shows that the points with higher w_i (in red) are close to the boundary between the two classes. Indeed, in classification, VBSW can be interpreted as a local label agreement. This behavior verifies recent findings of Xu et al. (2021) where authors conclude that in classification, a good set of weights would put importance on points close to the decision boundary.

We train a Multi-Layer Perceptron of 1 layer of 4 units, using Stochastic Gradient Descent (SGD) and binary cross-entropy loss function, on a 300 points training data set for 50 random seeds. In this experiment, VBSW, i.e. weighting the data set with w_i is compared to the baseline where no weights are applied. Figure 3.4 (b) and (d) displays the decision boundary of best fit for each method. VBSW provides a cleaner decision boundary than baseline. These pictures and the results of Table 3.3 show the improvement obtained with VBSW.

	VBSW	baseline
DM	**99.4**, **94.44** $\pm$ 0.78	99, 92.06 $\pm$ 0.66
BH	**13.31**, **13.38** $\pm$ 0.01	14.05, 14.06 $\pm$ 0.01
BC	**99.12**, 97.6 $\pm$ 0.34	98.25, 97.5 $\pm$ 0.11

Table 3.3: **best, mean + se** for each method. The metric used is accuracy for DM and BC and Mean Squared Error for BH.

For BH data set, a linear model is trained, and for BC data set, an MLP of 1 layer and 30 units, with a train-validation split of $80\% - 20\%$. Both models are trained with Adam Kingma and Ba (2015). Since these data sets are small and the models are light, we study the effects of m and k on the error. Moreover, BH is a regression task and BC a classification task, so it allows studying the effect of hyperparameters more extensively.

For BH and BC experiments, we conduct a grid search for VBSW on the values of m and k. As a reminder, m is the ratio between the highest and the lowest weights, and k is the number of neighbor points used to compute the local variance. We train a linear model for BH and a MLP with 30 units for BC with VBSW on a grid of 20 values of m equally distributed between 2 and 100 and 20 values of k equally distributed between 10 and 50. As a result, we train the model on 400 pairs of (m, k) values and with 10 different random seeds for each pair.

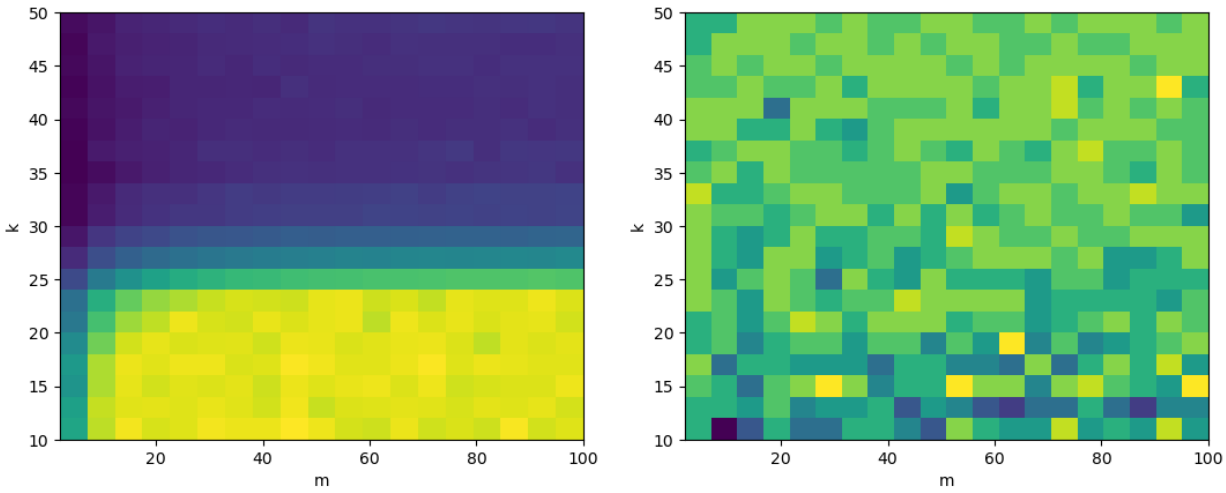


Figure 3.5: Color map of the error, with respect to m and k. **Left:** BH data set, for the mean of the MSE across 10 different seeds and **right:** BC data set, for the mean of $1 - acc$ across these seeds. Blue is lower.

These experiments, illustrated in Figure 3.5 show that the influence of m and k on the performances of the model can be different. For BH data set, low values of k clearly lead to poorer performances. Hyperparameter m seems to have less impact, although it should be chosen not too far from its lowest value, 2. For BC data set, on the contrary, the best performances are obtained for low values of k, while a high value could be chosen for m. These experiments highlight that the impact of m and k can be different between classification and regression, but it could also be different depending on the data set. Hence, we recommend considering these hyperparameters like many others involved in deep learning, selecting their values using hyperparameter optimization techniques. This procedure is illustrated in Chapter 6, where we use the hyperparameter optimization algorithm described in Chapter 4.

It also shows that many different (m, k) pairs lead to error improvement. It suggests that the weights approximation does not have to be exact for VBSW to be effective, as stated in Section 3.5.4.

3.4.3 Cost efficiency of VBSW

VBSW's computational burden mostly relies on the complexity of the nearest neighbor search algorithm, which is independent and can be used as a third-party algorithm. When the data set is not too large, classical techniques like KDtree (Bentley, 1975) can be used. However, when the number of points and the dimension of the data set increase, approximate nearest neighbors searches may be necessary to keep satisfying performances. In the previous examples, KDtree is more than sufficient. However, since we deal with more complex examples in the following, we directly use nmslib (Boytsov and Naidan, 2013), an approximate nearest neighbors search library for homogeneity of the implementation.

3.5 VBSW for deep learning

The high dimensionality of many deep learning problems makes VBSW difficult to apply in the form previously described. In this part, we adapt VBSW to such problems and study its application to various real-world learning tasks. We also study the robustness of VBSW and its complementarity with other similar techniques.

3.5.1 Methodology

We mentioned that local variance could be computed using already existing points. This statement implies finding the nearest neighbors of each point. In extremely high dimension spaces like image spaces, the curse of dimensionality makes nearest neighbors spurious. In addition, the data structure may be highly irregular, and the concept of nearest neighbor may be misleading. Thus, it would be irrelevant to evaluate $\widehat{D^2 f_{\boldsymbol{\epsilon}}}$ directly on this data.

One of the strengths of deep learning is to construct good representations of the data embedded in lower-dimensional latent spaces. For instance, in Computer Vision, convolutional neural networks' deeper layers represent more abstract features. We could leverage this representational power of neural networks and simply apply our methodology within this latent feature space.

Variance Based Samples Weighting (VBSW) for deep learning is recapitulated in Algorithm 3. Here, $\mathcal{M}$ is the initial neural network whose feature space will be used to project the training data set and apply VBSW. **Line 1:** m and k are hyperparameters that can be chosen jointly with all other hyperparameters, e.g. using a random search. Their effects and interactions are studied and discussed in Sections 3.4.2 and 3.5.4. **Line 2:** The initial neural network, $\mathcal{M}$, is trained as usual. Notations $\{(\frac{1}{N}, \boldsymbol{x}_1), ..., (\frac{1}{N}, \boldsymbol{x}_N)\}$ is equivalent to $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, because all the weights are the same $(\frac{1}{N})$. **Line 3:** The last fully connected layer is discarded, resulting in a new model $\mathcal{M}^*$, and the training data set is projected in the feature space. **Line 4-5:** equation (3.4) is applied to compute the weights w_i that are used to weight the projected data

set. **Line 6:** The last layer is re-trained (which is often equivalent to fitting a linear model) using the weighted data set and added to $\mathcal{M}^*$ to obtain the final model $\mathcal{M}_f$. As a result, $\mathcal{M}_f$ is a composition of the already trained model $\mathcal{M}^*$ and $f_{\boldsymbol{\theta}}$ trained using the weighted data set.

Algorithm 3 Variance Based Samples Weighting (VBSW) for deep learning

1: **Inputs:** k, m, $\mathcal{M}$
2: Train $\mathcal{M}$ on the training set $\{(\frac{1}{N}, \boldsymbol{x}_1), ..., (\frac{1}{N}, \boldsymbol{x}_N)\}$, $\{(\frac{1}{N}, f(\boldsymbol{x}_1)), ..., (\frac{1}{N}, f(\boldsymbol{x}_N))\}$
3: Construct $\mathcal{M}^*$ by removing its last layer
4: Compute $\{\widehat{Df^2}(\mathcal{M}^*(\boldsymbol{x}_1)), ..., \widehat{Df^2}(\mathcal{M}^*(\boldsymbol{x}_N))\}$ using equation (3.4).
5: Construct a new training data set $\{(w_1, \mathcal{M}^*(\boldsymbol{x}_1)), ..., (w_N, \mathcal{M}^*(\boldsymbol{x}_N))\}$
6: Train $f_{\boldsymbol{\theta}}$ on $\{(w_1, f(\boldsymbol{x}_1)), ..., (w_N, f(\boldsymbol{x}_N))\}$ and add it to $\mathcal{M}^*$. The final model is $\mathcal{M}_f = f_{\boldsymbol{\theta}} \circ \mathcal{M}^*$

3.5.2 Image Classification

In this section, we study the performances of VBSW on MNIST LeCun and Cortes (2010) and Cifar10 Krizhevsky et al. image classification data sets. For MNIST, we train LeNet (Lecun et al., 1998), with 40 different random seeds, and then apply VBSW for 10 different random seeds, with Adam optimizer and categorical cross-entropy loss. Note that in the following, Adam is used with the default parameters of its `keras` implementation. We record the best value obtained from the 10 VBSW training. We follow the same procedure for Cifar10, except that we train a ResNet20 for 50 random seeds and with data augmentation and learning rate decay. The networks have been trained on 4 Nvidia K80 GPUs. The values of the hyperparameters used can be found in **Appendix B**. We compare the test accuracy between LeNet 5 + VBSW, ResNet20 + VBSW, and the initial test accuracies of LeNet 5 and ResNet20 (baseline) for each of the initial networks.

	VBSW	baseline	gain per model
MNIST	**99.09**, **98.87** $\pm$ 0.01	98.99, 98.84 $\pm$ 0.01	**0.15**, **0.03** $\pm$ 0.01
Cifar10	**91.30**, **90.64** $\pm$ 0.07	91.01, 90.46 $\pm$ 0.10	**1.65**, **0.15** $\pm$ 0.04

Table 3.4: **best, mean + se** for each method. The metric used is accuracy. For a model $\mathcal{M}$, the gain g for this model is given by $g = \max_{1 \leq i \leq 10} (acc(\mathcal{M}_f^i) - acc(\mathcal{M}))$ with acc the accuracy and $\mathcal{M}_f^i$ the VBSW model trained at the i-th random seed.

The results statistics are gathered in Table 3.4, which also displays statistics about the gain due to VBSW for each model. The results on MNIST, for all statistics and for the gain, are significantly better than for the baseline. For Cifar10, we get a 0.3% accuracy improvement for the best model and up to 1.65% accuracy gain, meaning that among the 50 ResNet20s, there is one whose accuracy has been improved by 1.65% using VBSW. Note that applying VBSW took less than 15 minutes on a laptop with an i7-7700HQ CPU. A visualization of the samples weighted by the highest w_i is given in Figure 3.6.

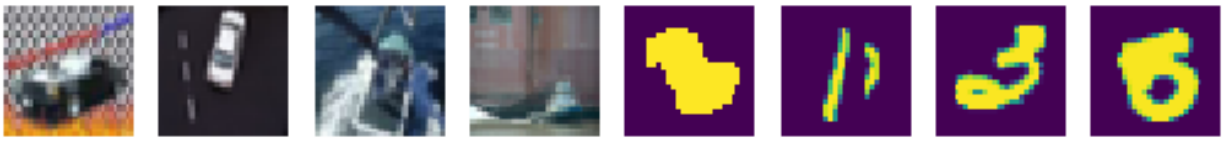


Figure 3.6: Samples from Cifar10 and MNIST with high w_i. Those pictures are either unusual or difficult to classify, even for a human (especially for MNIST).

3.5.3 Text Classification and Regression

In this section, we study the performances of VBSW on RTE and MRPC, two text classification data sets, and STS-B, a text classification data set, extracted from the glue benchmark Wang et al. (2019). For this application, we use Bert, a modern neural network based on transformers Vaswani et al. (2017b) that is the state-of-the-art of text-based machine learning tasks. We do not pre-train Bert, like in the previous experiments, since it has been originally built for Transfer Learning purposes. Therefore, its purpose is to be used as-is and then fine-tuned on any text data set see Devlin et al. (2019). However, because of the small size of the data set and the high number of model parameters, we chose not to fine-tune the Bert model and only to use the representations of the data sets in its feature space to apply VBSW. More specifically, we use tiny-bert Turc et al. (2019), which is a lighter version of the initial Bert. We train the linear model with TensorFlow to be able to add the trained model on top of the Bert model and obtain a unified model. RTE and MRPC are classification tasks, so we use binary cross-entropy loss function to train our models. STS-B is a regression task, so the model is trained with Mean Squared Error. All the models are trained with Adam optimizer. For each task, we compare the training of the linear model with VBSW and without VBSW (baseline). The results obtained with VBSW are better overall, except for Pearson Correlation in STS-B, which is slightly worse than baseline (Table 3.5).

	VBSW		baseline	
	m1	m2	m1	m2
RTE	**61.73**, **58.46** $\pm$ 0.15	-	61.01, 58.09 $\pm$ 0.13	-
STS-B	**62.31**, **62.20** $\pm$ 0.01	**60.99**, 60.88 $\pm$ 0.01	61.88, 61.87 $\pm$ 0.01	60.98, **60.92** $\pm$ 0.01
MRPC	**72.30**, **71.71** $\pm$ 0.03	**82.64**, **80.72** $\pm$ 0.05	71.56, 70.92 $\pm$ 0.03	81.41, 80.02 $\pm$ 0.07

Table 3.5: **best, mean + se** for each method. For RTE the metric used is accuracy (m1). For MRPC, metric 1 (m1) is accuracy and metric 2 (m2) is F1 score. For STS-B, metric 1 (m1) is Spearman correlation and metric 2 (m2) is Pearson correlation.

3.5.4 Robustness of VBSW

VBSW relies on statistical estimation: the weights are based on local empirical variance, evaluated using k points. In addition, they are re-scaled using hyperparameter m. Section

3.4.2 shows that many different combinations of m and k and, therefore, many different values for the weights improve the error. This behavior suggests that VBSW is quite robust to weights approximation error.

We also assess the robustness of VBSW to label noise. To that end, we train a ResNet20 on Cifar10 with four different noise levels. We randomly change the label of $p\%$ training points for four different values of p (10, 20, 30 and 40). We then apply VBSW 30 times and evaluate the obtained neural networks on a clean test set. The results are gathered in Table 3.6.

noise	10%	20%	30%	40%
original error	87.43	85.75	84.05	81.79
VBSW	**87.76**, 8**7.63** $\pm$ 0.01	**86.03**, **85.89** $\pm$ 0.01	**84.35**, **84.18** $\pm$ 0.02	**82.48**, **82.32** $\pm$ 0.02

Table 3.6: **best, mean + se** of the training of a ResNet20 on Cifar10 for different label noise levels. These results illustrate the robustness of VBSW to labels noise.

The results show that VBSW is still effective despite label noise. Since label noise essentially hurts weights approximation, this may be related to the robustness of VBSW to weights approximation error, described previously.

Although VBSW is robust to label noise, note that the goal of VBSW is not to address noisy label problem, like discussed in Section 3.1. It may be more effective to use a sampling technique tailored specifically for this situation.

3.5.5 Complementarity of VBSW

Existing techniques based on dataset processing can be used jointly with VBSW, by applying the first technique during the initial training of the neural network and then applying VBSW on its feature space. To illustrate this specificity, we compare VBSW with the recently introduced Active Bias (AB) (Chang et al., 2017) and transfer-learning-based curriculum learning (TCL) (Hacohen and Weinshall, 2019). AB dynamically weights the samples based on the variance of the probability of prediction of each point throughout the training, and TCL creates a curriculum based on sample difficulty evaluated on previously trained neural networks. Here, we study the effects of AB and TCL combined with VBSW for the training of a ResNet20 on Cifar10. Table 3.7 gathers the results of experiments for different baselines: vanilla, for regular training with Adam optimizer, AB / TCL for training with AB / TCL, VBSW for the application of VBSW on top of regular training, and VBSW + AB / VBSW + CL for initial training with AB / TCL and the application of VBSW. Unlike in Section 3.5.2, we do not use data augmentation nor learning rate decay in order to simplify the experiments.

The accuracy obtained with VBSW is quite similar to AB. While TCL yields better results than VBSW alone, the best accuracy is obtained when they are used jointly. Overall, the best neural networks are obtained when AB and TCL are used along with VBSW (AB + VBSW and TCL + VBSW), which demonstrates the complementarity of VBSW with other dataset

	accuracy (%)	VBSW gpm
vanilla	75.88, 74.55 ± 0.11	-
AB	76.33, 75.14 ± 0.09	-
TCL	78.54, 77.46 ± 0.07	-
VBSW	76.57, 74.94 ± 0.10	0.94, 0.40 ± 0.03
AB + VBSW	76.60, 75.33 ± 0.09	0.40, 0.14 ± 0.02
TCL + VBSW	**79.86**, **78.71** ± 0.09	**2.19**, **1.26** ± 0.08

Table 3.7: **Best, mean + se** of the training of 60 ResNet20s on Cifar10 for vanilla, VBSW, AB and AB + VBSW. The gain per model (gpm) g is defined by $g = \max_{1 \leq i \leq 10} (acc(\mathcal{M}_f^i) - acc(\mathcal{M}))$ with acc the accuracy and $\mathcal{M}_f^i$ the VBSW model trained at the i-th random seed.

processing techniques. Note that VBSW works much better when applied to a neural network initially trained with TCL. It means that TCL creates neural network features particularly suited to VBSW. This lead might be explored in future works.

3.6 Discussion and Perspectives

By studying the training distribution of the neural network, we explored a practical and classical question that naturally arises when performing surrogate modeling for approximating computer codes: how to construct the training set? This question comes from the methodology of designs of experiments, which is closely related to numerical and uncertainty analysis.

Nonetheless, this work emphasizes the interplay between numerical analysis and machine learning. Indeed, we found that exploring this question led to findings that are also relevant for approximation theory, which is an important component of machine learning. In the process of using supervised deep learning for numerical analysis, we contributed to supervised deep learning, and we did so by using principles from numerical analysis.

Hence, the results obtained in this chapter are impactful both for machine learning in numerical simulations and machine learning in general.

3.6.1 Impact for numerical simulations

This work comes from the observation that neural networks are more efficient when more data are sampled in the regions where the function to be learnt is steeper. It is an attempt to formalize this observation and to construct a workable methodology out of it. As a result, the methodologies for constructing the distribution $d\mathbb{P}_{\bar{\mathbf{x}}}$ can be used as new, principled designs of experiments.

In the context of numerical simulations, once $d\mathbb{P}_{\tilde{\mathbf{x}}}$ is constructed, it is possible to sample new data from it. It alleviates **Problem 2**, described in Section 3.3.2. In theory, **Problem 1** is also solved since we could have access to the derivatives - either by instrumenting the code with automatic differentiation if we have access to its implementation or by estimating them with finite differences. However, the implementation of automatic differentiation can be tedious, and if the computer code is slow and high dimensional, finite differences may be unaffordable. In that case, it is possible to use a third methodology based on the approximation of $\{Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_1), ..., Df^2_{\boldsymbol{\epsilon}}(\boldsymbol{x}_N)\}$ using local variance, like VBSW, and the sampling of new points, like TBS.

Finally, the method allows improving the error of neural networks without increasing the computational cost of their prediction. This achievement is of interest when they are intended to accelerate numerical simulations.

3.6.2 Impact for machine learning

VBSW is validated on several tasks, complementary with other training distribution modification frameworks, and robust to noise. It makes it quite versatile. Moreover, the problem of high dimensionality and irregularity of f, which often arises in deep learning problems, is alleviated by focusing on the latent space of neural networks. This makes VBSW scalable. As a result, VBSW can be applied to complex neural networks such as ResNet, or Bert, for various machine learning tasks.

The experiments support an original view of the learning problem that involves the local variations of f. The studies of Section 3.2.2, that use the derivatives of the function to be learnt to sample a more efficient training data set, support this approach as well. This view is also bolstered up by conclusions of Xu et al. (2021). VBSW allows extending this original view to problems where the derivatives of f are not accessible and sometimes not defined. Indeed, VBSW comes from Taylor expansion, which is specific to derivable functions, but in the end, it can be applied regardless of the properties of f.

Finally, this method is cost-effective. In most cases, it allows to quickly improve the performances of a neural network using a regular CPU. It is better than carrying on entirely new training with a wider and deeper neural network.

Chapter 4

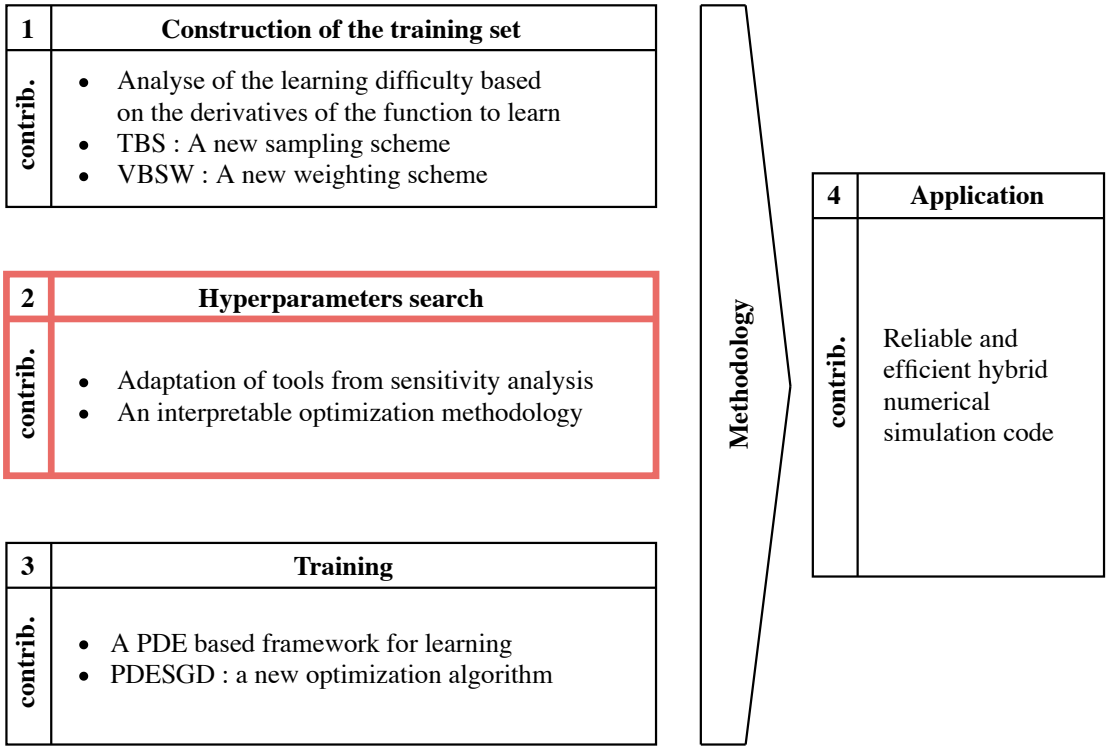
Hyperparameter optimization using goal-oriented sensitivity analysis

Hyperparameter optimization is ubiquitous in machine learning, and especially in deep learning, where neural networks are often cluttered with lots of hyperparameters. For applications to real-world machine learning tasks, finding good hyperparameters is mandatory, but can be fastidious for different reasons. i) The high number of hyperparameters by itself makes this problem challenging. ii) Their impact on error changes very often depending on the problem, so it is difficult to adopt general best practices and permanently recommend hyperparameter values for every machine learning problem. iii) Hyperparameters can be of very different natures, like continuous, discrete, categorical, or boolean, and have non-trivial relations, like conditionality or interactions. Besides, when deep learning is applied to accelerate numerical simulation, hyperparameter optimization suffers from one additional constraint: the obtained neural network has to be cost-effective.

In this chapter, we tackle these problems by proposing a sensitivity analysis applied to hyperparameter search space. To this end, we select a powerful metric used for goal-oriented sensitivity analysis, called Hilbert-Schmidt Independence Criterion (HSIC) (Gretton et al., 2005), which is a distribution dependence measure initially used for two-sample test problem (Gretton et al., 2007). Once adapted to hyperparameter search space, HSIC gives insights into hyperparameters' relative importance in a deep learning problem.

Using HSIC in hyperparameters space is non-trivial due to their complex structure. First, hyperparameters can be discrete (width of the neural network), continuous (learning rate), categorical (activation function), or boolean (batch normalization (Ioffe and Szegedy, 2015)). Second, some hyperparameter's presence is conditional to others (e.g. moments decay rates specific to Adam optimizer (Kingma and Ba, 2015)). Third, they can strongly interact (as shown in Tan and Le (2019): in some cases, it is better to increase depth and width by a similar factor). The metric should be able to compare hyperparameters reliably in such situations. We introduce solutions to overcome these obstacles and illustrate them with some

simple examples. Once adapted to such complex spaces, we show that HSIC allows us to understand hyperparameter relative importance better and to focus research efforts on specific hyperparameters. We also identify hyperparameters that have an impact on execution speed but not on the error. Then, we introduce ways of reducing the hyperparameter's range of possible values to improve the stability of the training and neural network's execution speed. Finally, we propose an HSIC-based optimization methodology in two steps, one focused on essential hyperparameters and the other on remaining hyperparameters. Its efficiency is validated on real-world problems: MNIST, Cifar10, and the approximation of the resolution of Bateman equations used in Section 3.2.2.



Methodology for supervised deep learning in numerical simulations

Contents

4.1 Sensitivity analysis as a new approach to hyperparameter optimization

In this section, we describe the challenges of hyperparameter optimization. We emphasize the limits of classical hyperparameter optimization algorithms used to tackle these challenges and legitimate the approach of sensitivity analysis.

4.1.1 Challenges of hyperparameter optimization

Let a neural network be described by n_h hyperparameters $\mathrm{x}_1, ..., \mathrm{x}_{n_h}$ with $\mathrm{x}_i \in \mathcal{X}_i$ and $\boldsymbol{\sigma} = (\mathrm{x}_1, ..., \mathrm{x}_{n_h})$. We denote $F(\boldsymbol{\sigma})$ the error of the neural network on a test data set once trained on a training data set. The aim of hyperparameter optimization is to find $\boldsymbol{\sigma}^* = \underset{\boldsymbol{\sigma}}{\operatorname{argmin}} F(\boldsymbol{\sigma})$. Even if its formulation is simple, neural networks hyperparameter optimization is a challenging task because of the great number of hyperparameters to optimize, the computational cost for evaluating $F(\boldsymbol{\sigma})$ and the complex structure of hyperparameter space. Figure 4.1 gives a graphical representation of a possible hyperparameter space and illustrates its complexity. Specific aspects to point out are the following ones :

- Hyperparameters do not live in the same measured space. Some are continuous (`weights_decay` $\in [10^{-6}, 10^{-1}]$), some are integers (`n_layers` $\in \{8, ..., 64\}$), others are categorical (`activation` $\in \{$`relu`$, ...,$ `sigmoid`$\}$), or boolean (`dropout` $\in \{True, False\}$).
- They could interact with each others. For instance `batch_size` adds variance on the objective function optimized by `optimizer`.
- Some hyperparameters are not involved for every neural networks configurations, e.g. `dropout_rate` is not used when `dropout = False` or `adam_beta` is only involved when `optimizer = adam`. In this case, we denote them as "conditional", otherwise we call them "main" hyperparameters.

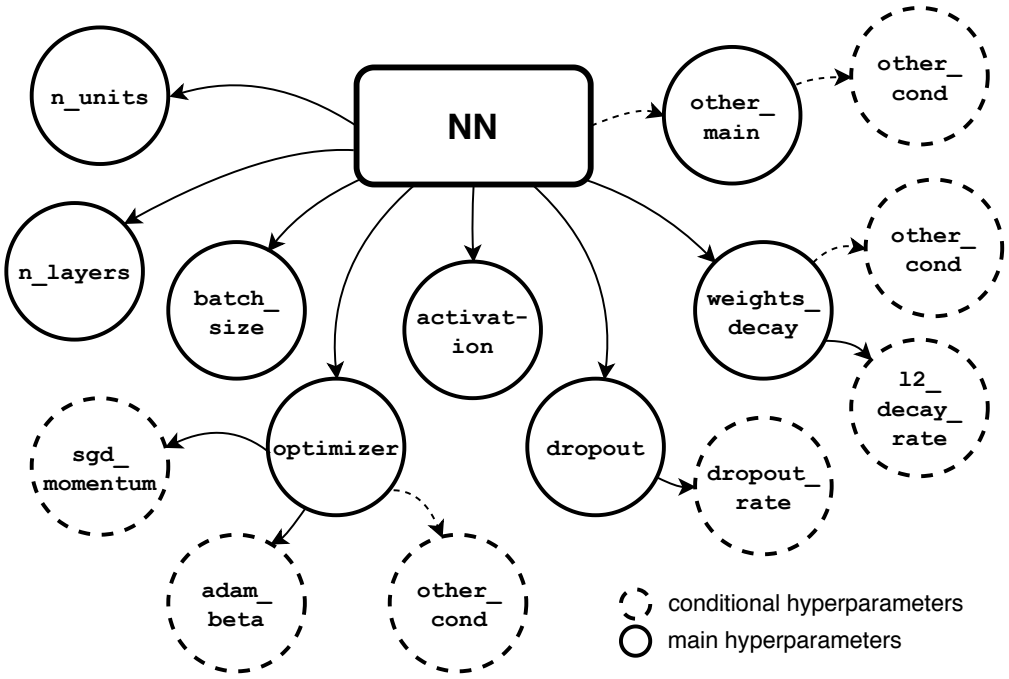


Figure 4.1: Example of hyperparameters space.

4.1.2 Classical hyperparameter optimization techniques

Many techniques have been introduced to tackle the problem of hyperparameter optimization. Grid search or random search (Bergstra and Bengio, 2012) uniformly explore the search space. The main difference between the two methods is that hyperparameters values are chosen on a uniform grid for a grid search. These values are deterministic, whereas, for a random search, hyperparameters values are randomly sampled from a uniform distribution in a Monte Carlo fashion. The main advantages of random search over grid search are that it allows for more efficient exploration of the hyperparameter search and that it is not constrained to a grid, so it does not suffer from the curse of dimensionality - which is a problem here since the hyperparameters can be pretty numerous. The standard costly part of these two methods is that it requires training a neural network for each hyperparameter configuration, so exploring the search space can be computationally very expensive.

Some methods aim at reducing the cost of such searches. For instance, Successive Halving (Jamieson and Talwalkar, 2016) and Hyperband (Li et al., 2018b) train neural networks in parallel, like in grid search or random search, and stop their training after a certain number of epochs. Then, they choose the best half of neural networks and carry on the training only for these neural networks, for the same number of epochs, and so on. This procedure allows testing more hyperparameters values for the same computational budget.

On the contrary, other methods are designed to improve the search quality with less training instances. Bayesian optimization, first introduced in Mockus (1974), is based on the approximation of the loss function by a surrogate model. After an initial uniform sampling

of hyperparameter configurations, the surrogate model is trained on these points and used to maximize an acquisition function. This acquisition function, often chosen to be expected improvement or upper confident bound (Shahriari et al., 2016), is supposed to lead to hyperparameter configurations that will improve the error. Therefore, it focuses the computation on potentially better hyperparameters values instead of randomly exploring the hyperparameters space. The surrogate model can be a Gaussian process (Snoek et al., 2012), a kernel density estimator (Bergstra et al., 2011) or even a neural network (Snoek et al., 2015).

Model-based hyperparameter optimization is not easily and naturally applicable to conditional or categorical hyperparameters that often appear when optimizing a neural network architecture. Such categorical hyperparameter can be the type of convolution layer for a convolutional neural network, regular convolution or depth-wise convolution (Chollet, 2016); and a conditional hyperparameter could be the specific parameters of each different convolution type. Neural architecture search explicitly tackles this problem. It dates back to evolutionary and genetic algorithms (Eiben and Schoenauer, 2002; Stanley and Miikkulainen, 2002) and has been the subject of many recent works. For instance, Kandasamy et al. (2018) models the architecture as a graph, or Pham et al. (2018); Tan et al. (2018) use reinforcement learning to automatically construct representations of the search space. See Elsken et al. (2019) for an exhaustive survey of this field. Nevertheless, their implementation can be tedious, often involving numerous hyperparameters themselves.

Classical hyperparameter optimization methods handle hyperparameter optimization quite successfully. However, they are end-to-end algorithms that return one single neural network. The user does not interact with the algorithm during its execution. This lack of interactivity has many automating advantages but can bring some drawbacks. First, these methods do not give any insight on the relative importance of hyperparameters, whereas it may be of interest in the first approach to a machine learning problem. They are black boxes and not interpretable. Second, one could have other goals than the accuracy of a neural network, like execution speed or memory consumption. Some works like Tan et al. (2018) introduce multi-objective hyperparameter optimization, but it requires additional tuning of the hyperparameter optimization algorithm itself. Finally, there may be flaws in the hyperparameters space, like a useless hyperparameter that could be dropped but is included in the search space and becomes a nuisance for the optimization. This aspect is all the more problematic since some popular algorithms, like gaussian process-based Bayesian optimization, suffer from the curse of dimensionality. We can sum up the drawbacks as lack of interpretability, difficulties in a multi-objective context, and unnecessary search space complexity.

4.1.3 Benefits of Sensitivity Analysis

In this work, we alleviate these concerns by mixing hyperparameter optimization with hyperparameter analysis. In other words, we construct an approach to hyperparameter optimization that relies on assessing hyperparameter's effects on the neural network's performances.

One powerful tool to analyze the effect of some input variables on the variability of a quantity

of interest is sensitivity analysis (Razavi et al., 2021). Sensitivity analysis consists of studying the sensitivity of the output of a function to its inputs. We could define this function as F and its inputs as $\boldsymbol{\sigma}$. Then, it would be possible to make hyperparameter optimization benefit from characteristics of sensitivity analysis. Indeed, sensitivity analysis allows specifically:

- Analyzing the relative importance of input variables for explaining the output, which helps to answer the lack of interpretability problem. We could explain and better understand hyperparameters' impact on the neural network error.
- Selecting practically convenient values for input variables with a limited negative impact on the output. It simplifies the multi-objective approach since we could, for instance, select values that improve execution speed with a limited impact on the neural network error.
- Identifying where to efficiently put research efforts to improve the output, which answers the unnecessary search space complexity problem. Indeed, we could focus on fewer hyperparameters to optimize by knowing which of them most impact the neural network error.

4.1.4 Goal-oriented sensitivity analysis

Several types of sensitivity measures can be estimated after an initial sampling of input vectors and their corresponding output values. The first type of metric gives information about the contribution of an input variable to the output based on variance analysis. The most common metric used for that purpose are Sobol indices (Sobol, 1993), but they only assess the contribution of variables to the output variance. Goal-oriented Sobol indices (Fort et al., 2016) or uncertainty importance measure (Borgonovo, 2007) construct quantities based on the output whose variance analysis gives more detailed information. However, computing these indices can be very costly since estimating them with an error of $\mathcal{O}(\frac{1}{\sqrt{n_s}})$ requires $(n_h+2)\times n_s$ sample evaluations (Saltelli, 2002; Prieur and Tarantola, 2016), which can be prohibitive for hyperparameter analysis. Another type of metrics, called dependence measures, assesses the dependence between x_i (that can be a random vector) and the output $F(\boldsymbol{\sigma})$ (Da Veiga, 2013). It relies on the claim that the more x_i is independent of $F(\boldsymbol{\sigma})$, the less important it is to explain it. Dependence measures are based on dissimilarity measures between $\mathbb{P}_{\mathrm{x}_i}\mathbb{P}_{\mathrm{y}}$ and $\mathbb{P}_{\mathrm{x}_i,\mathrm{y}}$, where $\mathrm{x}_i \sim \mathbb{P}_{\mathrm{x}_i}$ and $\mathrm{y} = F(\boldsymbol{\sigma}) \sim \mathbb{P}_{\mathrm{y}}$, since $\mathbb{P}_{\mathrm{x}_i\mathrm{y}} = \mathbb{P}_{\mathrm{x}_i}\mathbb{P}_{\mathrm{y}}$ when x_i and y are independent. In Da Veiga (2013), the author gives several examples of indices based on dissimilarity measures like f-divergences (Csizar, 1967) or integral probability metrics (Müller, 1997). These indices are easier and less expensive to estimate (n_s training instances instead of $(n_h+2)\times n_s$) than variance-based measures since they only need a simple Monte Carlo design of experiment.

This work focuses on a specific dependence measure, known as the Hilbert-Schmidt Independence Criterion (HSIC). The following sections are dedicated to the description of HSIC and its adaptation to hyperparameter optimization.

4.2 HSIC-based goal oriented sensitivity analysis

In this section, we recall the definition of Hilbert Schmidt Independence Criterion, how to use it in practice, and how to adapt it in order to perform goal-oriented sensitivity analysis.

4.2.1 From Integral Probability Metrics to Maximum Mean Discrepancy

Let $\mathbf{x}$ and $\mathbf{y}$ be two random variables of probability distribution $\mathbb{P}_{\mathbf{x}}$ and $\mathbb{P}_{\mathbf{y}}$ defined in $\mathcal{X}$. Gretton et al. (2007) show that distributions $\mathbb{P}_{\mathbf{x}} = \mathbb{P}_{\mathbf{y}}$ if and only if $\mathbb{E}[f(\mathbf{x})] - \mathbb{E}[f(\mathbf{y})] = 0$ for all $f \in C(\mathcal{X})$, where $C(\mathcal{X})$ is the space of bounded continuous functions on $\mathcal{X}$. This lemma explains the intuition behind the construction of Integral Probability Metrics (IPM) (Müller, 1997).

Let $\mathcal{F}$ be a class of functions, $f : \mathcal{X} \to \mathbb{R}$. An IPM γ is defined as

$$\gamma(\mathcal{F}, \mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}}) = \sup_{f \in \mathcal{F}} |\mathbb{E}[f(\mathbf{x})] - \mathbb{E}[f(\mathbf{y})]|. \tag{4.1}$$

The Maximum Mean Discrepancy (MMD) can be defined as an IPM restricted to a class of functions $\mathcal{F}_{\mathcal{H}}$ defined on the unit ball of a Reproducing Kernel Hilbert Space (RKHS) $\mathcal{H}$ of kernel $k : \mathcal{X}^2 \to \mathbb{R}$. In Gretton et al. (2005), this choice is motivated by the capacity of RKHS to embed probability distributions efficiently. The authors define $\mu_{\mathbf{x}}$ such that $\mathbb{E}(f(\mathbf{x})) = \langle f, \mu_{\mathbf{x}} \rangle_{\mathcal{H}}$ as the mean embedding of $\mathbb{P}_{\mathbf{x}}$. Then, $\gamma_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ can be written

$$\begin{aligned} \gamma_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}}) &= \|\mu_{\mathbf{x}} - \mu_{\mathbf{y}}\|_{\mathcal{H}}^2. \\ &= \int\int k(\boldsymbol{x}, \boldsymbol{x})(p_{\mathbf{x}}(\boldsymbol{x}) - p_{\mathbf{y}}(\boldsymbol{x}))(p_{\mathbf{x}}(\boldsymbol{y}) - p_{\mathbf{y}}(\boldsymbol{y}))d\boldsymbol{x}d\boldsymbol{x}' \\ &= \mathbb{E}[k(\mathbf{x}, \mathbf{x}')] + \mathbb{E}[k(\mathbf{y}, \mathbf{y}')] - 2\mathbb{E}[k(\mathbf{x}, \mathbf{y})], \end{aligned} \tag{4.2}$$

where $p_{\mathbf{x}}(\boldsymbol{x})dx = d\mathbb{P}_{\mathbf{x}}$, $p_{\mathbf{y}}(\boldsymbol{y})dy = d\mathbb{P}_{\mathbf{y}}$, $\mathbf{x}, \mathbf{x}'$ are iid (independent and identically distributed) and $\mathbf{y}, \mathbf{y}'$ are iid. After a Monte Carlo sampling of $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_{n_s}\}$ and $\{\boldsymbol{y}_1, ..., \boldsymbol{y}_{n_s}\}$, $\gamma_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ can thus be estimated by $\hat{\gamma}_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$, with

$$\hat{\gamma}_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}}) = \sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k(\boldsymbol{x}_j, \boldsymbol{x}_l) + \sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k(\boldsymbol{y}_j, \boldsymbol{y}_l) - 2\sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k(\boldsymbol{x}_j, \boldsymbol{y}_l), \tag{4.3}$$

and $\hat{\gamma}_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ being an unbiased estimator, its standard error can be estimated using the empirical variance of $\hat{\gamma}_k^2(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$.

4.2.2 The kernel choice

Equation (4.2) involves to choose a kernel k. In practice, k is chosen among a class of kernels that depends on a set of parameters $\mathbf{h} \in \mathbf{H}$, where $\mathbf{H}$ is a kernel parameter space. We therefore temporarily denote the kernel by $k_{\mathbf{h}}$. Examples of kernels are the Gaussian Radial Basis Function $k_h : (\boldsymbol{x}, \boldsymbol{y}) \to \exp(-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2h^2})$ or the Matérn function $k_{\mathbf{h}} : (\boldsymbol{x}, \boldsymbol{y}) \to \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{||\boldsymbol{x}-\boldsymbol{y}||}{\eta}\right)^{\nu} K_\nu \left(\sqrt{2\nu}\frac{||\boldsymbol{x}-\boldsymbol{y}||}{\eta}\right)$, where $\mathbf{h} = \{\sigma, \nu, \eta\}$, Γ is the gamma function and K_ν is the modified Bessel function of the second kind. In Fukumizu et al. (2009), the authors study the choice of the kernel, and more importantly of the kernel parameters $\mathbf{h}$. They state that, for the comparison of probabilities $\mathbb{P}_{\mathbf{x}}$ and $\mathbb{P}_{\mathbf{y}}$, the final parameter $\mathbf{h}^*$ should be chosen such that

$$\gamma^2_{k_{\mathbf{h}^*}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}}) = \sup_{\mathbf{h}\in\mathbf{H}} \gamma^2_{k_{\mathbf{h}}}(\mathbb{P}_{\mathbf{y}}, \mathbb{P}_{\mathbf{y}}). \tag{4.4}$$

The authors suggest focusing on unnormalized kernel families, like Gaussian Radial Basis Functions $\left\{k_h : (\boldsymbol{x}, \boldsymbol{y}) \to \exp(-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2h^2}), h \in (0, \infty)\right\}$, also used in Da Veiga (2013), for which they demonstrate that $\hat{\gamma}^2_{k_{\mathbf{h}^*}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$, defined as

$$\hat{\gamma}^2_{k_{\mathbf{h}^*}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}}) = \sup_{\mathbf{h}\in\mathbf{H}} \left[\sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k_{\mathbf{h}}(\boldsymbol{x}_j, \boldsymbol{x}_l) + \sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k_{\mathbf{h}}(\boldsymbol{y}_j, \boldsymbol{y}_l) - 2\sum_{j=1}^{n_s}\sum_{l=1}^{n_s} k_{\mathbf{h}}(\boldsymbol{x}_j, \boldsymbol{y}_l) \right], \tag{4.5}$$

is a consistent estimator of $\gamma^2_{k_{\mathbf{h}^*}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$. It is thus possible to choose $\mathbf{h}$ by maximizing $\hat{\gamma}^2_{k_{\mathbf{h}}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ with respect to $\mathbf{h}$. Therefore, in this work, we use Gaussian Radial Basis Functions kernel. Once $\mathbf{h}^*$ is chosen, the approximation error of $\hat{\gamma}^2_{k_{\mathbf{h}^*}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ can also be estimated like in Section 4.2.1. It is important to note that $\gamma^2_{k_{\mathbf{h}}}(\mathbb{P}_{\mathbf{x}}, \mathbb{P}_{\mathbf{y}})$ can be estimated in a $\mathcal{O}(n_s^2)$ computational complexity, which is not expensive given usual values of n_s in hyperparameter optimization context. The total complexity of the minimization process depends on the minimization algorithm, but since the optimization problem is low dimensional - there is never more than a handful of kernel parameters - the whole process is always cost effective. To simplify the notations, we denote $k_{\mathbf{h}^*}$ by k in the following sections.

4.2.3 Hilbert-Schmidt Independence Criterion (HSIC) for goal-oriented sensitivity analysis

Let $\mathbf{x} \in \mathcal{X}$ and $\mathbf{y} \in \mathcal{Y}$, and $\mathcal{G}$ the RKHS of kernel $k : \mathcal{X}^2 \times \mathcal{Y}^2 \to \mathbb{R}$. HSIC can be written

$$HSIC(\mathbf{x}, \mathbf{y}) = \gamma^2_k(\mathbb{P}_{\mathbf{xy}}, \mathbb{P}_{\mathbf{x}}\mathbb{P}_{\mathbf{y}}) = \|\mu_{\mathbf{xy}} - \mu_{\mathbf{x}}\mu_{\mathbf{y}}\|_{\mathcal{G}}. \tag{4.6}$$

Then, HSIC measures the distance between $\mathbb{P}_{\mathbf{xy}}$ and $\mathbb{P}_{\mathbf{y}}\mathbb{P}_{\mathbf{x}}$ embedded in $\mathcal{H}$. Indeed, since

$\mathbf{x} \perp \mathbf{y} \Rightarrow \mathbb{P}_{\mathbf{xy}} = \mathbb{P}_{\mathbf{y}}\mathbb{P}_{\mathbf{x}}$, the closer these distributions are, in the sense of γ_k, the more independent they are.

In Spagnol et al. (2018), the authors present a goal oriented sensitivity analysis by focusing on the sensitivity of F w.r.t. $\mathbf{x}_i$ when $\mathrm{y} = F(\mathbf{x}_1, ..., \mathbf{x}_{n_h}) \in \mathbf{Y}$, with $\mathbf{Y} \subset \mathbb{R}$. The sub-space $\mathbf{Y}$ is chosen based on the goal of the analysis. In the context of optimization, for instance, $\mathbf{Y}$ is typically chosen to be the best percentile of Y. To achieve this, the authors introduce a new random variable, $\mathrm{z} = \mathbf{1}_{\mathrm{y}\in\mathbf{Y}}$. Then,

$$HSIC(\mathbf{x}_i, \mathrm{z}) = \mathbb{P}(\mathrm{z} = 1)^2 \times \gamma_k^2(\mathbb{P}_{\mathbf{x}_i|\mathrm{z}=1}, \mathbb{P}_{\mathbf{x}_i}), \tag{4.7}$$

so $HSIC(\mathbf{x}_i, \mathrm{z})$ measures the distance between $\mathbf{x}_i$ and $\mathbf{x}_i|\mathrm{z} = 1$ (to be read $\mathbf{x}_i$ conditioned to $\mathrm{z} = 1$) and can be used to measure the importance of $\mathbf{x}_i$ to reach the sub-space $\mathbf{Y}$ with F. Using the expression of γ_k given by equation (4.2), its exact expression is

$$HSIC(\mathbf{x}_i, \mathrm{z}) = \mathbb{P}(\mathrm{z} = 1)^2 \Big[\mathbb{E}[k(\mathbf{x}_i, \mathbf{x}_i')] + \mathbb{E}[k(\mathbf{z}, \mathbf{z}')] - 2\mathbb{E}[k(\mathbf{x}_i, \mathbf{z})]\Big], \tag{4.8}$$

where $\mathbf{x}_i, \mathbf{x}_i'$ are iid and $\mathbf{z}, \mathbf{z}'$ are iid.It is estimated for each $\mathbf{x}_i$ using Monte Carlo estimators denoted by $S_{\mathbf{x}_i,\mathbf{Y}}$, based on samples $\{\boldsymbol{x}_{i,1}, ..., \boldsymbol{x}_{i,n_s}\}$ from $\mathbf{x}_i \sim d\mathbb{P}_{\mathbf{x}_i}$ and corresponding $\{z_1, ..., z_{n_s}\}$ drawn from z. the estimator $S_{\mathbf{x}_i,\mathbf{Y}}$ is defined as

$$\begin{aligned} S_{\mathbf{x}_i,\mathbf{Y}} = \mathbb{P}(\mathrm{z} = 1)^2 \Bigg[& \frac{1}{m^2} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(\boldsymbol{x}_{i,j}, \boldsymbol{x}_{i,l}) \delta(z_j = 1) \delta(z_l = 1) \\ & + \frac{1}{{n_s}^2} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(\boldsymbol{x}_{i,j}, \boldsymbol{x}_{i,l}) \\ & - \frac{2}{n_s m} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(\boldsymbol{x}_{i,j}, \boldsymbol{x}_{i,l}) \delta(z_l = 1) \Bigg], \end{aligned} \tag{4.9}$$

with $m = \sum_k^{n_s} \delta(z_k = 1)$ and $\delta(a) = 1$ if a is True and 0 otherwise. We use this metric in the following. This section mainly summed up the mathematics on which the sensitivity indices are based and how they are used in practice in a sensitivity analysis context. The following section is devoted to the application of HSIC in hyperparameter space.

4.3 Application of HSIC to hyperparameters space

HSIC has two advantages that make it stand out from other sensitivity indices and make it particularly suitable to the hyperparameters space. First, equation (4.9) emphasizes that it is

possible to estimate HSIC using simple Monte Carlo estimation. Hence, in the context of hyperparameter optimization, such indices can be estimated after a classical random search. Secondly, using equation (4.7), HSIC allows to perform goal-oriented sensitivity analysis easily, i.e. to assess the importance of each hyperparameter for the error to reach a given $\mathbf{Y}$. For hyperparameter analysis, $\mathbf{Y}$ can be chosen to be the sub-space for which $F(\mathrm{x}_1, ..., \mathrm{x}_{n_h})$ is in the best percentile p of a metric (L_2 error, accuracy,...), say $p = 10\%$. Then, the quantity $S_{\mathrm{x}_i, \mathbf{Y}}$ measures the importance of each hyperparameter x_i for obtaining the 10% best neural networks.

However, one cannot use HSIC as is in hyperparameter analysis. Indeed, hyperparameters do not live in the same measured space, they could interact with each other, and some are not directly involved for each configuration. In the following sections, we suggest some original solutions to these issues.

To illustrate the performances of these solutions, we consider a toy example which is the approximation of Runge function $r : x \to \frac{1}{1+15x^2}$, $x \in [-1, 1]$ by a fully connected neural network. This approximation problem is a historical benchmark of approximation theory. We consider $n_h = 14$ different hyperparameters (see **Appendix B** for details). We randomly draw $n_s = 10000$ hyperparameter configurations and perform the corresponding training on 11 training points. We record the test error on a test set of 1000 points. All samples are equally spaced between 0 and 1.

In Section 4.3.1, we introduce a transformation to deal with hyperparameters that do not live in the same measured space. Then, in Section 4.3.2 we explain how to use HSIC to evaluate hyperparameters' interactions. Finally, in Section 4.3.3 we deal with conditionality between hyperparameters.

4.3.1 Normalization of hyperparameters space

Hyperparameters can be defined in very different spaces. For instance, the activation function is a categorical variable that can be `relu`, `sigmoid` or `tanh`, `dropout_rate` is a continuous variable between 0 and 1 while `batch_size` is an integer that can go from 1 to hundreds. Moreover, it may be useful to sample hyperparameters with a non-uniform distribution (e.g. log-uniform for `learning_rate`). Doing so affects HSIC value and its interpretation, which is undesirable since this distribution choice is arbitrary and only relies on practical considerations. Let us illustrate this phenomenon in the following example.

Example Let $f : [0, 2]^2 \to \{0, 1\}$ such that

$$f(\mathrm{x}_1, \mathrm{x}_2) = \begin{cases} 1 & \text{if } \mathrm{x}_1 \in [0, 1], \mathrm{x}_2 \in [0, 1], \\ 0 & \text{otherwise.} \end{cases}$$

Suppose we want to assess the importance of x_1 and x_2 for reaching the goal $f(\mathrm{x}_1, \mathrm{x}_2) = 1$ without knowing f. In the formalism of the previous section, we have $\mathbf{Y} = \{1\}$. Regarding its definition, x_1 and x_2 are equally important for f to reach $\mathbf{Y}$, due to their symmetrical effect. Let $\mathrm{x}_1 \sim \mathcal{N}(1, 0.1, [0, 2])$ (normal distribution of mean 1 and variance 0.1 truncated between 0 and 2) and $\mathrm{x}_2 \sim \mathcal{U}[0, 2]$. We compute $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{\mathrm{x}_2,\mathbf{Y}}$ with $n_s = 10000$ points and display their value in Table 4.1. The values of $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{\mathrm{x}_2,\mathbf{Y}}$ are quite different. It is natural since their chosen initial distribution is different. However, this choice has nothing to do with their actual importance; it should not have any impact on the importance measure. Here, we could erroneously conclude that x_2 is more important than x_1.

	x_1	x_2
$S_{\mathrm{x},\mathbf{Y}}(\times 10^{-2})$	1.17 ± 0.05	1.55 ± 0.05

Table 4.1: $S_{\mathrm{x},\mathbf{Y}}$ values for x_1 and x_2

	u_1	u_2
$S_{\mathrm{x},\mathbf{Y}}(\times 10^{-2})$	1.54 ± 0.05	1.55 ± 0.05

Table 4.2: $S_{\mathrm{x},\mathbf{Y}}$ values for u_1 and u_2

This example shows that we have to make $S_{\mathrm{x}_i,\mathbf{Y}}$ and $S_{\mathrm{x}_j,\mathbf{Y}}$ comparable in order to say that hyperparameter x_i is more important than hyperparameter x_j. Indeed, if x_i and x_j do not follow the same distribution or $\mathcal{X}_i \neq \mathcal{X}_j$, it may be irrelevant to compare them directly. We need a method to obtain values for $S_{\mathrm{x}_i,\mathbf{Y}}$ that are robust to the choice of $d\mathbb{P}_{\mathrm{x}_i}$. To tackle this problem, we introduce the following approach for comparing variables with HSIC. Let Φ_i be the CDF of x_i. We have that $\Phi_i(\mathrm{x}_i) = \mathrm{u}_i$, with $\mathrm{u}_i \sim \mathcal{U}[0, 1]$. After an initial Monte Carlo sampling of hyperparameter x_i, which can be a random search, we can apply Φ_i to each input point to obtain u_i corresponding to x_i with u_i iid, so living in the same measured space. Yet, one must be aware that to obtain $\mathrm{u}_i \sim \mathcal{U}[0, 1]$, its application is different for continuous and discrete variables:

- for continuous variables, $\Phi_i(\mathrm{x}_i)$ is a bijection between $\mathcal{X}_i$ and $[0, 1]$ so Φ_i can be applied on draws from x_i.
- For categorical, integer or boolean variables, $\Phi_i(\mathrm{x}_i)$ is not a bijection between $\mathcal{X}_i$ and $[0, 1]$. Suppose that x_i is a discrete variable with p possible values $\{\mathrm{x}_i[1], ..., \mathrm{x}_i[p]\}$, each with probability w_p. Let us encode $\{\mathrm{x}_i[1], ..., \mathrm{x}_i[p]\}$ by $\{1, ..., p\}$. Then, $\Phi_i(\mathrm{x}_i) = \sum_{j=1}^{p} w_j \mathbf{1}_{[\mathrm{x}_i \leq j]}(\mathrm{x}_i)$. When Φ_i is applied as is, $\Phi_i(\mathrm{x}_i)$ is not uniform. To overcome that, one can simply use $\mathrm{u}_i = \sum_{j=1}^{p} \mathcal{U}[\sum_{k<j} w_k, \sum_{k<j+1} w_k]\delta(\mathrm{x}_i = j)$. This trick, introduced in Kruskal (1964), is commonly used in Monte Carlo resolution of Partial Differential Equations (Gillespie, 1976). As a result, $\mathrm{u}_i \sim \mathcal{U}[0, 1]$.

Finally, all we have to do is sampling x_i, like in a classical random search, following the distribution we want, and then apply Φ_i to obtain u_i. The corresponding HSIC estimation is $S_{\mathrm{u}_i,\mathbf{Y}}$. It only involves u_i and $\mathrm{u}_i|\mathrm{z} = 1$ and since u_i are iid, the comparison of different $S_{\mathrm{u}_i,\mathbf{Y}}$ becomes relevant. Coming back to the previous example, Table 4.2 displays values of $S_{\mathrm{u}_1,\mathbf{Y}}$ and $S_{\mathrm{u}_2,\mathbf{Y}}$. This time, the value is the same, leading to the correct conclusion that both variables are equally important. Note that in the following, we denote $S_{\mathrm{u}_i,\mathbf{Y}}$ by $S_{\mathrm{x}_i,\mathbf{Y}}$ for clarity but always resort to this transformation.

Let us apply this methodology to the Runge approximation hyperparameter analysis problem. Figure 4.2 displays a comparison between $S_{\mathrm{x}_i,\mathbf{Y}}$ for hyperparameters of the Runge approximation problem, with $\mathbf{Y}$ the set of the 10% best neural networks. For readability, we order x_i by $S_{\mathrm{x}_i,\mathbf{Y}}$ value in the legend and the figure. We also display black error bars corresponding to HSIC estimation standard error. This graphic highlights that `optimizer` is by far the most important hyperparameter for this problem, followed by `activation`, `loss_function` and `n_layers`. Other hyperparameters may be considered non-impactful because their $S_{\mathrm{x}_i,\mathbf{Y}}$ values are low. Besides, these values are lower than the error evaluation. It could be only noise, and therefore these hyperparameters can not be ordered on this basis.

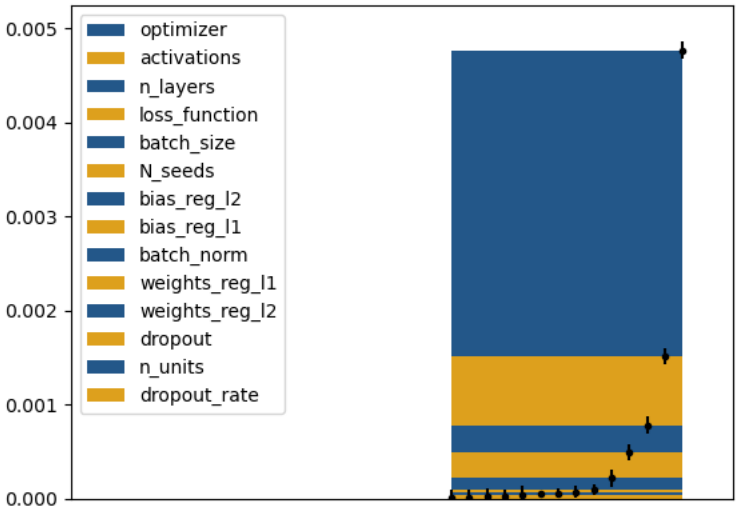


Figure 4.2: Comparison of $S_{\mathrm{x}_i,\mathbf{Y}}$ for hyperparameters in Runge approximation problem. The hyperparameters are ordered from the most important (top of the legend) to the least important (bottom of the legend), and their value is graphically represented in a stacked bar plot following the same order.

4.3.2 Interactions between hyperparameters

If $S_{\mathrm{x}_i,\mathbf{Y}}$ is low, it means that $\mathbb{P}_{\mathrm{x}_i}$ and $\mathbb{P}_{\mathrm{z}}$ are similar (in the sense of HSIC). We could want to conclude that x_i has a limited impact on Y. However, x_i may have an impact due to its interactions with the other hyperparameters. In other words, let x_i and x_j be two variables, it can happen that $S_{\mathrm{x}_i,\mathbf{Y}}$ and $S_{\mathrm{x}_j,\mathbf{Y}}$ are low while $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ is high.

Example For instance let $f : [0,2]^3 \to \{0,1\}$ such that

$$f(\mathrm{x}_1,\mathrm{x}_2,\mathrm{x}_3) = \begin{cases} 1 & \text{if } \mathrm{x}_1 \in [0,1], \mathrm{x}_2 \in [1,2], \mathrm{x}_3 \in [0,1], \\ 1 & \text{if } \mathrm{x}_1 \in [0,1], \mathrm{x}_2 \in [0,1], \mathrm{x}_3 \in [1,2], \\ 0 & \text{otherwise.} \end{cases}$$

In that case, let $\mathbf{Y} = \{1\}$, $\forall x \in [0,2]$ we have $p_{\mathrm{x}_2|\mathrm{z}=1}(x) = p_{\mathrm{x}_2}(x)$ and $p_{\mathrm{x}_3|\mathrm{z}=1}(x) = p_{\mathrm{x}_3}(x)$. Hence, according to equation 4.8 we have $HSIC(\mathrm{x}_2, \mathrm{z}) = HSIC(\mathrm{x}_3, \mathrm{z}) = 0$. However, we have

$$\begin{aligned} HSIC(\mathrm{x}_1, \mathrm{z}) &= \mathbb{P}(\mathrm{z}=1)^2 \int_{[0,2]^2} k(x,x') \left[p_{\mathrm{x}_1|\mathrm{z}=1}(x) - p_{\mathrm{x}_1}(x)\right] \\ &\quad \times \left[p_{\mathrm{x}_1|\mathrm{z}=1}(x') - p_{\mathrm{x}_1}(x')\right] dx dx' \\ &= \frac{1}{8}\Big[\int_{[0,1]\times[0,1]} k(x,x') dx dx' + \int_{[1,2]\times[1,2]} k(x,x') dx dx' \\ &\quad - 2 \int_{[0,1]\times[1,2]} k(x,x') dx dx' \Big], \end{aligned}$$

so for non-trivial choice of k, $HSIC(\mathrm{x}_1, \mathrm{z}) \neq 0$. One could deduce that x_1 is the only relevant variable for reaching $\mathbf{Y}$, but in practice it is necessary to chose x_2 and x_3 carefully as well. For instance, if $\mathrm{x}_1 \in [0,1]$, $f(\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3) = 1$ if $\mathrm{x}_2 \in [1,2]$ and $\mathrm{x}_3 \in [0,1]$ but $f(\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3) = 0$ if $\mathrm{x}_2 \in [1,2]$ and $\mathrm{x}_3 \in [1,2]$. This is illustrated in Figure 4.3, which displays the histograms of x_1 and $\mathrm{x}_1|\mathrm{z}=1$, x_2 and $\mathrm{x}_2|\mathrm{z}=1$, x_3 and $\mathrm{x}_3|\mathrm{z}=1$, obtained from 10000 points $(\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3)$ sampled uniformly in the definition domain of f.

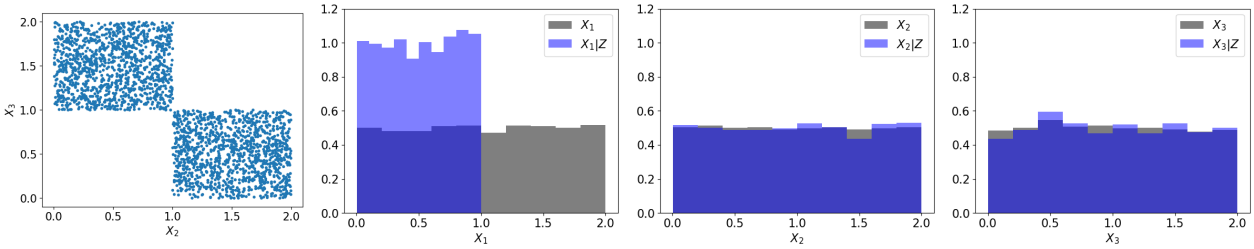


Figure 4.3: **From left to right: 1** - Pairs of $(\mathrm{x}_2|\mathrm{z}=1, \mathrm{x}_3|\mathrm{z}=1)$. **2** - Histogram of x_1 and $\mathrm{x}_1|\mathrm{z}=1$. **3** - Histogram of x_2 and $\mathrm{x}_2|\mathrm{z}=1$. **4** - Histogram of x_3 and $\mathrm{x}_3|\mathrm{z}=1$.

Histograms are the same for x_2, $\mathrm{x}_2|\mathrm{z}=1$ and x_3, $\mathrm{x}_3|\mathrm{z}=1$ (uniform between 0 and 2), but different for x_1, $\mathrm{x}_1|\mathrm{z}=1$. Therefore, HSIC being a distance measure between x_1 and $\mathrm{x}_1|\mathrm{z}=1$, it becomes intuitive that it will be high for x_1 and close to zero for x_2 and x_3, even if x_2 and x_3 are important as well because of their interaction. To assess this intuition, we compute $S_{\mathrm{x}_1,\mathbf{Y}}$, $S_{\mathrm{x}_2,\mathbf{Y}}$, $S_{\mathrm{x}_3,\mathbf{Y}}$ and $S_{(\mathrm{x}_2,\mathrm{x}_3),\mathbf{Y}}$ after simulating f for $n_s = 2000$ points. We also compute $S_{(\mathrm{x}_4,\mathrm{x}_5),\mathbf{Y}}$, with x_4 and x_5 two dummy variables, uniformly distributed, to have a reference for $S_{(\mathrm{x}_2,\mathrm{x}_3),\mathbf{Y}}$. The results can be found in Table 4.3. They show that $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{(\mathrm{x}_2,\mathrm{x}_3),\mathbf{Y}}$ are of the same order while $S_{\mathrm{x}_2,\mathbf{Y}}$, $S_{\mathrm{x}_3,\mathbf{Y}}$ and $S_{(\mathrm{x}_4,\mathrm{x}_5),\mathbf{Y}}$ are two decades lower than $S_{\mathrm{x}_1,\mathbf{Y}}$, which confirms that $S_{\mathrm{x},\mathbf{Y}}$ may be low while interactions are impactful.

	x_1	x_2	x_3	$(\mathrm{x}_2, \mathrm{x}_3)$	$(\mathrm{x}_4, \mathrm{x}_5)$
$S_{\mathrm{x},\mathbf{Y}}$	1.51×10^{-2}	6.26×10^{-6}	1.64×10^{-5}	3.47×10^{-3}	3.70×10^{-6}

Table 4.3: $S_{\mathrm{x},\mathbf{Y}}$ values for variables of the experiment

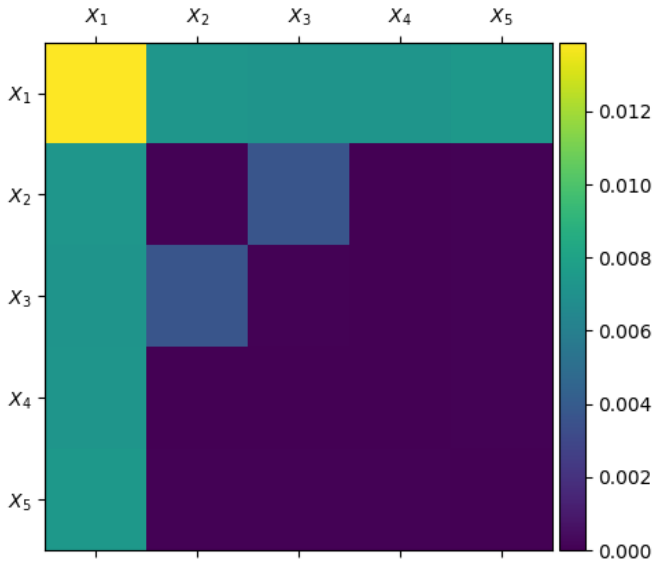
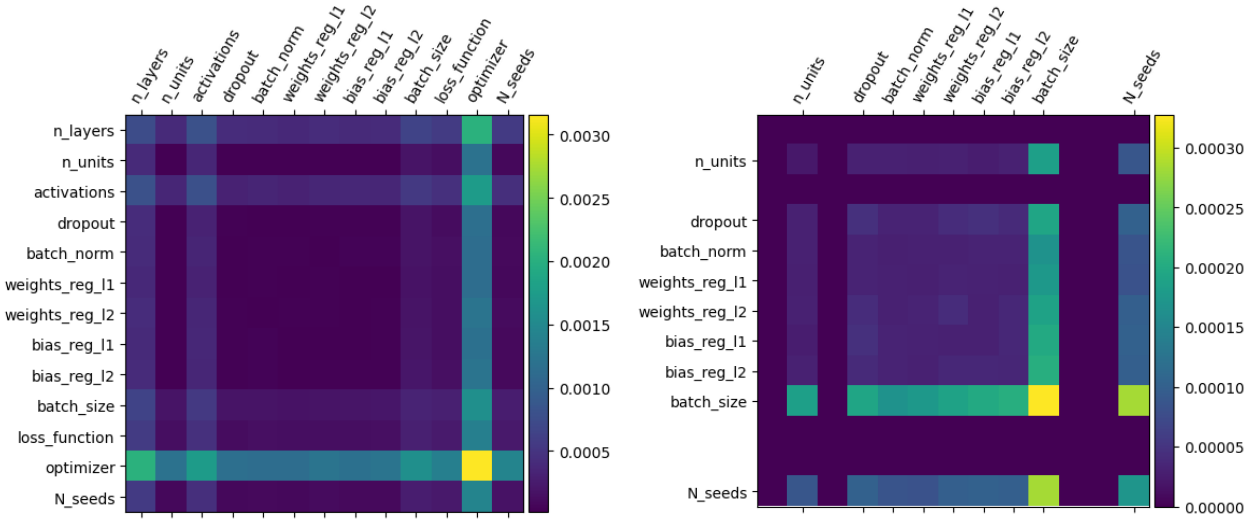


Figure 4.4: $S_{\mathrm{x},\mathbf{Y}}$ for each pair of variable.

Additionally, we display the $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ for each pair of variable x_i and x_j on Figure 4.4. We can see that for variables other than x_1, $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ is high only for $i = 2$ and $j = 3$. This example shows that it is necessary to compute $S_{\mathrm{x},\mathbf{Y}}$ of joint variables to perceive the importance of interactions between variables.

The values are easy to interpret in this example because we know the behavior of the underlying function f. In practice, $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{(\mathrm{x}_2,\mathrm{x}_3),\mathbf{Y}}$ can not be compared because $(\mathrm{x}_2, \mathrm{x}_3)$ and x_1 do not live in the same measured space ($\mathcal{X}_2 \times \mathcal{X}_3$ and $\mathcal{X}_1$ respectively). Moreover, like we see on Figure 4.4, $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ is always the highest when $i = 1$, regardless of j. In fact, if for a given variable x_i, $S_{\mathrm{x}_i,\mathbf{Y}}$ is high, so will be $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ for any other variable x_j. Hence, care must be taken to only compare interactions of low $S_{\mathrm{x},\mathbf{Y}}$ variables with each others, and not with high $S_{\mathrm{x},\mathbf{Y}}$ variables. Coming back to Runge approximation example, Figure 4.5a displays the $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ for each pair of hyperparameters, and Figure 4.5b for each pair of hyperparameters, except for the impactful hyperparameters `optimizer`, `activation`, `n_layers` and `loss_function`.

Figures 4.5a and 4.5b illustrate the remarks of the previous section. First, if we only look for interactions on Figure 4.5a, we would conclude that the most impactful hyperparameters are the only one to interact, and that they only interact with each others. Figure 4.5b shows that this conclusion is not true. Hyperparameter `batch_size` is the 5-th most impactful hyperparameter, and like we can see in Figure 4.2, is slightly above the remaining hyperparameters. It is normal that $S_{(\texttt{batch\_size},\mathrm{x}_j),\mathbf{Y}}$ is high, with x_j every other hyperparameters. However, $S_{(\texttt{batch\_size},\texttt{n\_units}),\mathbf{Y}}$ is higher, whereas `n_units` is the 13-th most impactful hyperparameter. This means that `batch_size` interacts with `n_units` in this problem, i.e. that when considered together, they contribute to explain the best results.



(a) (b)

Figure 4.5: (a) $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ for each pair of hyperparameters. (b) $S_{(\mathrm{x}_i,\mathrm{x}_j),\mathbf{Y}}$ for each pair of hyperparameters, except for `optimizer`, `activation`, `n_layers` and `loss_function`. The grid can be read symmetrically with respect to the diagonal.

4.3.3 Conditionality between hyperparameters

Conditionality between hyperparameters, which often arises in Deep Learning, is a non-trivial challenge in hyperparameter optimization. For instance, hyperparameter "`dropout_rate`" will only be involved when hyperparameter "`dropout`" is set to `True`. Classically, two approaches can be considered. The first **(i)** splits the hyperparameter optimization between disjoint groups of hyperparameters that are always involved together, like in Bergstra et al. (2011). Then, two separate instances of hyperparameter optimization are created, one for the main hyperparameters and another for `dropout_rate`. The second **(ii)** considers these hyperparameters as if they were always involved, even if they are not, like in Falkner et al. (2018). In that case, `dropout_rate` is always assigned a value even when `dropout` = `False`, and these dummy values are used in the optimization. First, we explain why these two approaches are not suited to our case. Then we propose a third approach **(iii)**.

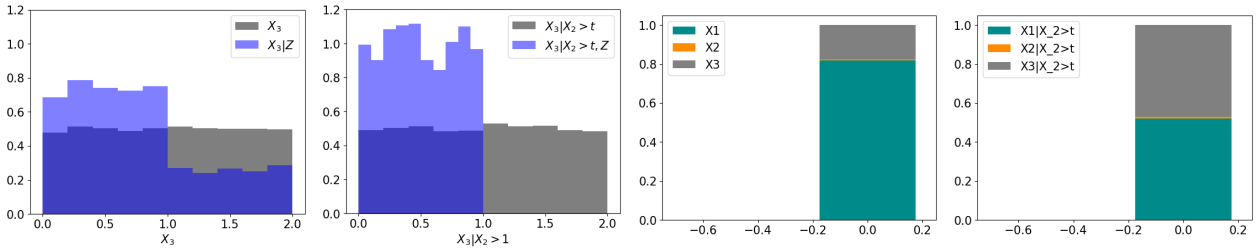
(i) The first formulation splits the hyperparameters between disjoints sets of hyperparameters whose value and presence are involved jointly in a training instance. In Runge approximation hyperparameter analysis, since `dropout_rate` is the only conditional hyperparameter, it would mean to split the hyperparameters between two groups: {`dropout_rate`} and another containing all the others. This splitting approach is not suited to HSIC computation because it produces disjoints sets of hyperparameters, while we would want to measure the importance of every hyperparameter and compare it to each other hyperparameter. Here, `dropout_rate` could not be compared to any other hyperparameters.

(ii) In the second case, if we apply HSIC with the same idea, we could compute HSIC of a hyperparameter with irrelevant values coming from configurations where it is not involved. Two situations can occur. First, if a conditional variable x_i is never involved in the hyperparameter configurations that yield the p-percent best accuracies (depending on the percentile chosen), the values used for computing $S_{\mathrm{x}_i,\mathbf{Y}}$, i.e. $\mathrm{x}_i|\mathrm{z}=1$, are drawn from the initial, uniform distribution u_i. Then, $S_{\mathrm{x}_i,\mathbf{Y}}$ will be very low, and the conclusion will be that it is not impactful for reaching the percentile, which is correct since none of the best neural networks have used this hyperparameter. However, if x_i is only involved in a subset of all tested hyperparameter configurations and is impactful in that case, $S_{\mathrm{x}_i,\mathbf{Y}}$ would be lowered by the presence of the other artificial values of x_i drawn from the uniform distribution. In that case, we could miss its actual impact. The following example illustrates this phenomenon.

Example. Let $f : [0,2]^3 \to \{0,1\}$ such that:

$$f(\mathrm{x}_1,\mathrm{x}_2,\mathrm{x}_3) = \begin{cases} B & \text{if } \mathrm{x}_1 \in [0,1], \mathrm{x}_2 \in [0,t] \\ 1 & \text{if } \mathrm{x}_1 \in [0,1], \mathrm{x}_2 \in [t,2], \mathrm{x}_3 \in [0,1], \\ 0 & \text{otherwise,} \end{cases}$$

With B a Bernoulli variable of parameter 0.5 and $t \in [0,2]$ (so that $S_{\mathrm{x}_2,\mathbf{Y}}$ is low). Let $\mathbf{Y} = \{1\}$. In that case, x_1 plays a key role for reaching $\mathbf{Y}$, and x_3 is taken into account only when $\mathrm{x}_2 > t$. In these cases, it is as important as x_1 for reaching $\mathbf{Y}$ and we would like to retrieve this information. Parameter t allows controlling how many values of x_3 will be involved. We evaluate f on $n_s = 2000$ points uniformly distributed across $[0,2]^3$, first with $t = 1$.



(a) (b) (c) (d)

Figure 4.6: **(a)** - Histogram of x_3 and $\mathrm{x}_3|\mathrm{z}=1$ **(b)** - Histogram of x_3 and $\mathrm{x}_3|\mathrm{x}_2 > t, \mathrm{z}$. **(c)** - $S_{\mathrm{x},\mathbf{Y}}$ for x_1; x_2 and x_3. **(d)** - $S_{\mathrm{x},\mathbf{Y}}$ for $\mathrm{x}_1|\mathrm{x}_2 > t$; $\mathrm{x}_2|\mathrm{x}_2 > t$ and $\mathrm{x}_3|\mathrm{x}_2 > t$.

Figure 4.6a compares the histograms of x_3 and $\mathrm{x}_3|\mathrm{z}=1$. Figure 4.6b compares histograms of $\mathrm{x}_3|\mathrm{x}_2 > t$ and of $\mathrm{x}_3|\mathrm{x}_2 > t, \mathrm{z}$. This shows that the distribution of $\mathrm{x}_3|\mathrm{z}=1$ is different if we choose to consider artificial values of x_3 or values of x_3 that are actually used by f ($\mathrm{x}_3|\mathrm{x}_2 > t$). Figures 4.6c and 4.6d show that relative values of $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{\mathrm{x}_3,\mathbf{Y}}$ are quite different whether we chose to consider $\mathrm{x}_2 > t$ or not, meaning that the conclusions about the impact of x_3 can be potentially different. To emphasize how different these conclusions can be, we compare $S_{\mathrm{x}_1,\mathbf{Y}}$ and $S_{\mathrm{x}_3,\mathbf{Y}}$ for different values of t. The results are displayed on Figure 4.7 (top row).

Since the value of t controls how much artificial values there are for x_3, this demonstrates how different $S_{x_3,\mathbf{Y}}$ can be, depending on the amount of artificial points. This experiment emphasizes the problem because in all cases, x_3 is equally important for reaching $\mathbf{Y}$ whereas for $t = 1.8$ we would be tempted to discard x_3.

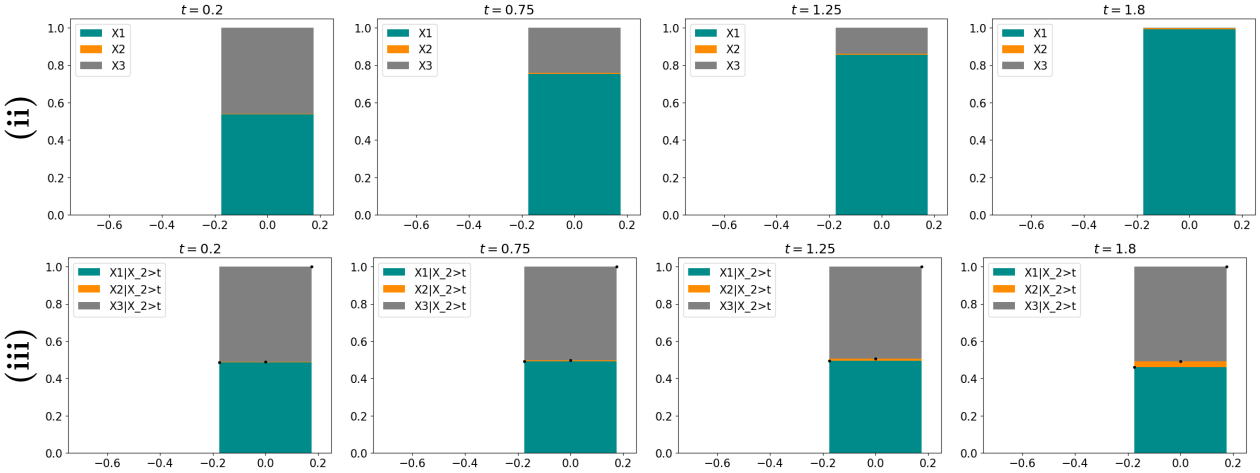
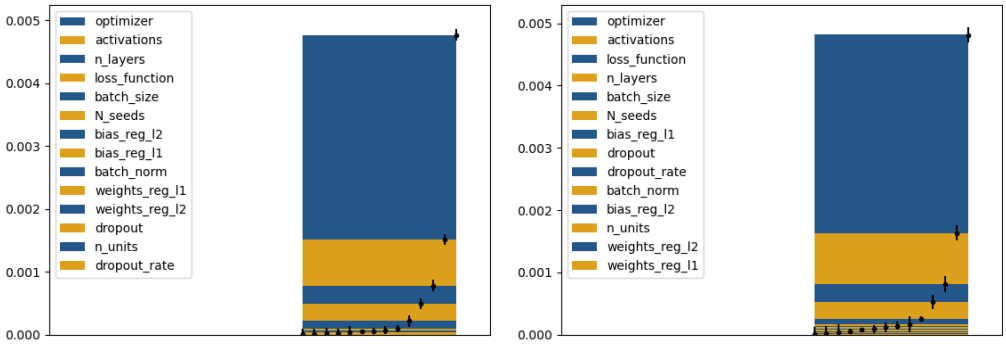


Figure 4.7: **Top (ii):** $S_{x,\mathbf{Y}}$ for x_1, x_2 and x_3 for different values of t. **Bottom (iii):** $S_{x,\mathbf{Y}}$ for $x_1|x_2 > t$, $x_2|x_2 > t$ and $x_3|x_2 > t$ for different values of t.

To sum up, this formulation brings significant implementation advantages because it allows computing $S_{x_i,\mathbf{Y}}$ as if there were no conditionality. However, it carries a risk to miss essential impacts of conditional hyperparameters and discard them illegitimately.

(iii) In this work, we propose a splitting strategy that produces sets of hyperparameters that are involved together in the training, but are not disjoints, unlike **(i)**. Let $\mathcal{J}_k \in \{1, ..., n_h\}$ be the set of indices of hyperparameters that can be involved in a training jointly with conditional hyperparameter x_k. We define $\mathcal{G}_{x_k} = \{x_i|x_k, i \in \mathcal{J}_k\}$, the set of hyperparameters involved jointly in hyperparameter configurations when x_k is also involved. By convention, we denote the set of all main hyperparameters by $\mathcal{G}_0$. In Runge problem, `dropout_rate` is the only conditional hyperparameter, so we have two sets $\mathcal{G}_0 = \{x_1, ..., x_{n_h}\} \setminus$ `dropout_rate` and $\mathcal{G}_{\texttt{dropout\_rate}} = \{x_1|\texttt{dropout\_rate}, ..., x_{n_h}|\texttt{dropout\_rate}\} = \{x_1|\texttt{dropout} = \texttt{true}, ..., x_{n_h}|\texttt{dropout} = \texttt{true}\}$. It is then possible to compute $S_{x_i,\mathbf{Y}}$ for $x_i \in \mathcal{G}_0$, identify the most impactful main hyperparameters, then to compute $S_{x_i,\mathbf{Y}}$ for $x_i \in \mathcal{G}_{\texttt{dropout\_rate}}$ and to assess if `dropout_rate` is impactful by comparing it to other variables of $\mathcal{G}_{\texttt{dropout\_rate}}$. On the example problem, we can compute $S_{x_i,\mathbf{Y}}$ only for x_1, x_2 and x_3 when $x_2 > t$. This set would be $\mathcal{G}_{x_3}$ (except that x_2 is not categorical nor integer - but in that case we can consider $\bar{X}_2 = \mathbf{1}(x_2 > t)$). On the bottom row of Figure 4.7, $S_{x_1|x_2>t,\mathbf{Y}}$ and $S_{x_3|x_2>t,\mathbf{Y}}$ keep approximately the same values for all t, which is the correct conclusion since when x_3 is involved (i.e. $x_2 > t$) , it is as important as x_1 for reaching $\mathbf{Y}$. Coming back to Runge, Figure 4.8 displays $S_{x_i,\mathbf{Y}}$ for Runge approximation for $x_i \in \mathcal{G}_{\texttt{dropout\_rate}}$, compared to the first approach where we do not care about conditionality, though in this specific case it does not change much of the conclusion that `dropout_rate` is not impactful.



(a) (b)

Figure 4.8: Comparison of $S_{\mathrm{x}_i,\mathbf{Y}}$ for variables $\mathrm{x}_i \in \mathcal{G}_0$ (a) and for variables $\mathrm{x}_i \in \mathcal{G}_{\texttt{dropout\_rate}}$ (b)

In Runge example, we have only considered one conditional hyperparameter, which is `dropout_rate`, leading to only two groups $\mathcal{G}_0$ and $\mathcal{G}_{\texttt{dropout\_rate}}$. For another, more complex example, we could introduce additional conditional hyperparameters such as SGD's `momentum`. In that case, there would be two additional groups. The group $\mathcal{G}_{\texttt{momentum}}$, that contains hyperparameters conditioned to when `momentum` is involved, but also $\mathcal{G}_{(\texttt{dropout\_rate},\texttt{momentum})}$ that contains hyperparameters conditioned to when `momentum` and `dropout_rate` are *simultaneously* involved. If the initial random search contains n_s configurations, `dropout_rate` and `momentum` are involved in $n_s/2$ configurations. HSIC estimation of hyperparameters of the groups $\mathcal{G}_{\texttt{dropout\_rate}}$ and $\mathcal{G}_{\texttt{momentum}}$ will be coarser but still acceptable. However, `dropout_rate` and `momentum` would only be involved simultaneously in $n_s/4$ configurations, which may lead to too inaccurate HSIC estimation for $\mathcal{G}_{(\texttt{dropout\_rate},\texttt{momentum})}$. This happens because `dropout_rate` and `momentum` do not depend on the same main hyperparameter. Hence, to avoid this problem, we only consider groups $\mathcal{G}$ with conditional hyperparameters that depend on the same main hyperparameter. In our case, these groups are $\mathcal{G}_0$, $\mathcal{G}_{\texttt{dropout\_rate}}$ and $\mathcal{G}_{\texttt{momentum}}$.

4.3.4 Summary: evaluation of HSIC in hyperparameter analysis

In this section, we summarize the results of the previous discussions to provide a methodology for evaluating the HSIC of hyperparameters in complex search spaces in Algorithm 4.

Comments on Algorithm 4. **Line 1:** one can choose any initial distribution for hyperparameters. **Line 2:** this step is a classical random search. Recall that HSIC evaluation can be applied after any random search, even if it was not initially conducted for HSIC estimation. Configurations $\boldsymbol{\sigma}_i$ are sampled from $\boldsymbol{\sigma} = (\mathrm{x}_1, ..., \mathrm{x}_{n_h}) \in \mathcal{H}$. **Line 3:** this step strongly benefits from parallelism. **Line 4:** the set $\mathbf{Y}$ is often taken as the p % percentile of $\{Y_1, ..., Y_{n_s}\}$, but can be any other set depending on what we want to assess. **Line 6 - 10:** the evaluation starts with main hyperparameters because they are always involved. Once most impactful main hyperparameters are selected, we assess the conditional ones.

Remark. *The value of* $S_{\mathrm{x}_i,\mathbf{Y}}$ *strongly depends on the initial distribution chosen for* x_i*. Indeed,*

Algorithm 4 Evaluation of HSIC in hyperparameter analysis

Inputs: hyperparameter search space $\mathcal{H} = \mathcal{X}_1 \times ... \times \mathcal{X}_{n_h}$, n_s.
Sample n_s hyperparameter configurations $\{\boldsymbol{\sigma}_1, ..., \boldsymbol{\sigma}_{n_s}\}$.
Train a neural network for each configuration and gather outputs $\{Y_1, ..., Y_{n_s}\}$.
Define $\mathbf{Y}$.
Construct conditional groups $\mathcal{G}_0, ...$
for each group, starting with $\mathcal{G}_0$ **do**
    Construct u_i for every x_i using Φ_i of section 4.3.1.
    Compute $S_{\mathrm{x}_i,\mathbf{Y}} := S_{\mathrm{u}_i,\mathbf{Y}}$ using equation (4.9).
    By comparing them, select the most impactful hyperparameters.
    Check for interacting hyperparameters.
Outputs: Most impactful hyperparameters and interacting hyperparameters.

if the distribution only spans values of x_i *that yield good prediction error,* $S_{\mathrm{x}_i,\mathbf{Y}}$ *will be low. Conversely, if it spans good values but also includes absurd values,* $S_{\mathrm{x}_i,\mathbf{Y}}$ *will be higher. Hence, without a priori knowledge, we recommend to select a large range of values for each* x_i

4.4 Experiments

Now that we can compute and correctly assess HSIC, we introduce possible usages of this metric in the context of hyperparameter analysis. In this section, we explore three benefits that we can draw from HSIC based hyperparameter analysis.

- Interpretability: HSIC allows analyzing hyperparameters, obtaining knowledge about their relative impact on error.
- Stability: Some hyperparameter configurations can lead to dramatically high errors. A hyperparameters range reduction based on HSIC can prevent such situations.
- Acceleration: We can choose values for less important hyperparameters that improve inference and training time.

We illustrate these points through hyperparameter analysis when training a fully connected neural network on MNIST and a convolutional neural network on Cifar10. We also study the approximation by a fully connected neural network of Bateman equations solution, in a more complex way than in Section 3.2.2. Details about the construction of Bateman equations data set can be found in **Appendix C** and hyperparameters space and conditional groups $\mathcal{G}_0, ...$ for each problem in **Appendix B**.

4.4.1 Hyperparameter analysis

This section presents a first analysis of the estimated value of HSIC for the three benchmark data sets: MNIST, Cifar10, and Bateman equations. These evaluations are based on an initial random search for $n_s = 1000$ different hyperparameter configurations. The set $\mathbf{Y}$ is the 10%-best errors percentile, so n_s is taken sufficiently large for HSIC to be correctly estimated. Indeed, if $n_s = 1000$, there will be 100 samples of $\mathrm{u}_i|\mathrm{z} = 1$. For every data set, we extract 10% of the training data to construct a validation set to evaluate z. We keep a test set for evaluating neural networks obtained after hyperparameter optimization described in Section 4.5. This random search was conducted using 100 parallel jobs on CPU nodes for fully connected neural networks and 24 parallel jobs on Nvidia Tesla V100 and P100 GPUs for convolutional neural networks, so the results for these configurations were obtained quite quickly, in less than two days.

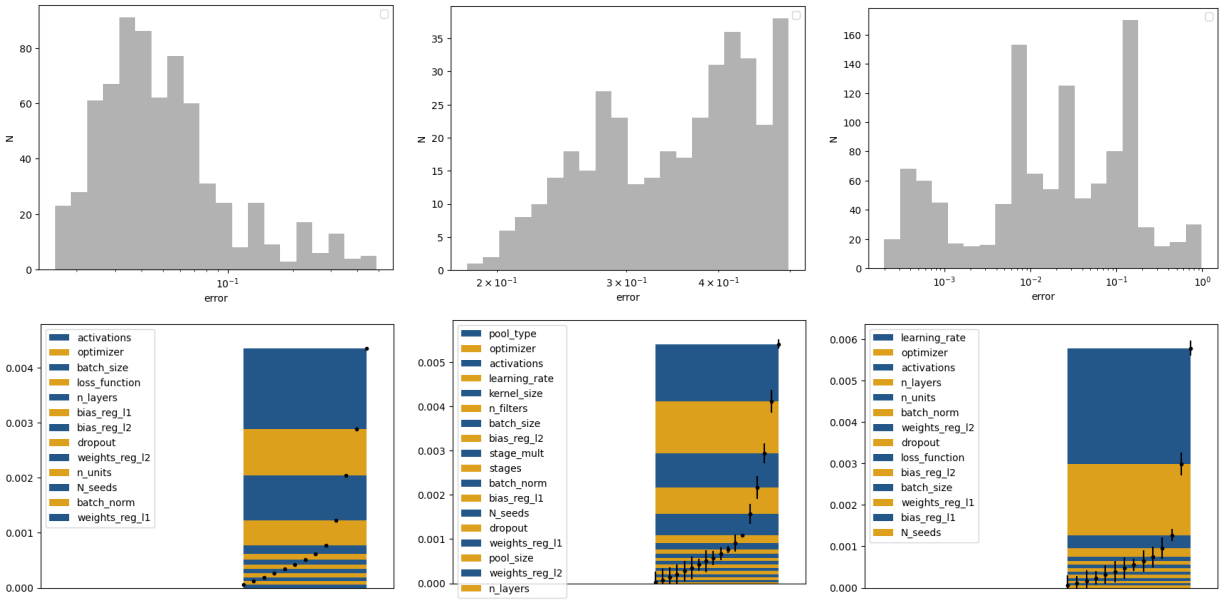
Note that for each data set, graphical comparison of $S_{\mathrm{x}_i,\mathbf{Y}}$ for conditional groups $\mathcal{G}_0, ...$ is displayed in **Appendix D**, for conciseness and clarity.

4.4.1.1 MNIST

We train $n_s = 1000$ different neural networks. We can see on Figure 4.9a that the accuracy goes up to $\sim 99\%(1 - \text{error})$ which is quite high for a fully connected neural network on MNIST. Figure 4.9a also displays the values of $S_{\mathrm{x}_i,\mathbf{Y}}$ for each hyperparameter x_i stacked vertically. Here, `activation`, `optimizer`, `batch_size` and `loss_function` have significantly high $S_{\mathrm{x}_i,\mathbf{Y}}$. Hyperparameter `n_layers` also stands out from the remaining hyperparameter, while staying far below `loss_function` HSIC. There is one conditional group to consider, $\mathcal{G}_{\texttt{dropout\_rate}}$, and `dropout_rate` is found not to be impactful.

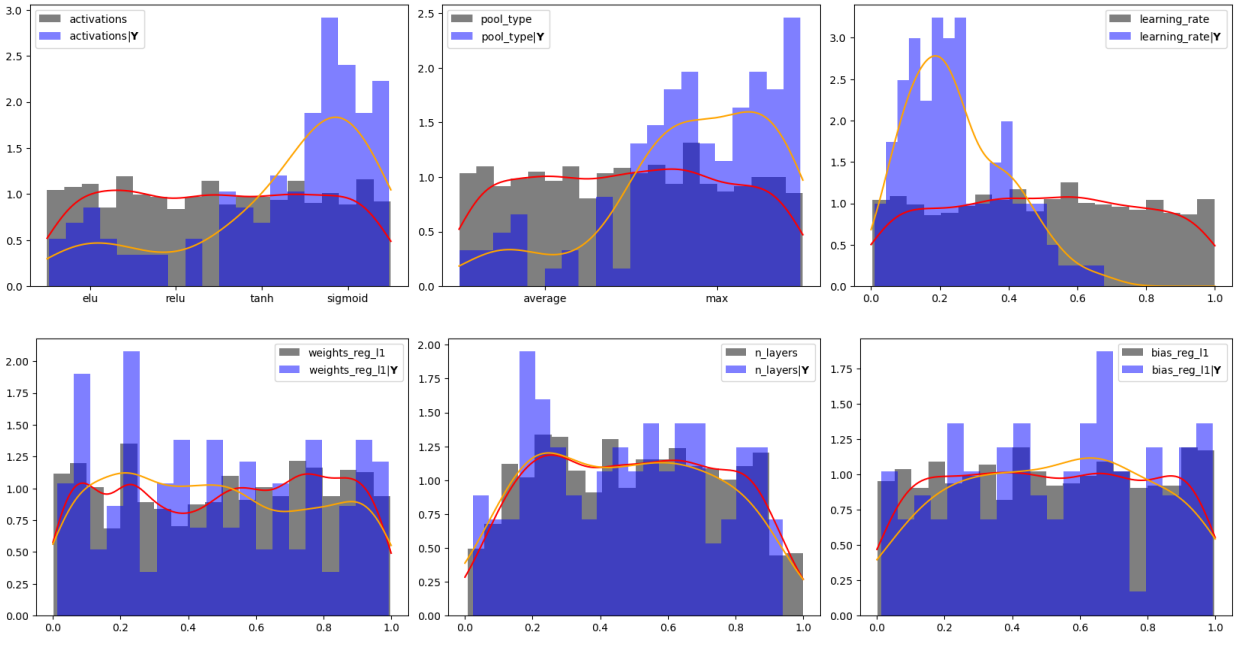
Interestingly, neither the depth (`n_layers`) nor the width (`n_units`) are among the most important hyperparameters. Notice that the random search yields a neural network of depth 4 and width 340 which obtained 98.70% accuracy, while the best networks (there were two) obtained 98.82% accuracy for a depth of 10 and a width of 791 and 1403, respectively. Recall that the min-max depth was 1-10 and width was 134-1500. It means that lighter networks are capable of obtaining competitive accuracy. Another interesting observation is that `loss_function` does not have the highest HSIC, meaning that Mean Squared Error allows obtaining good test errors, which is surprising for a classification problem.

We plot histograms of u_i and $\mathrm{u}_i|\mathrm{z} = 1$ on Figure 4.10a for `activation` (top) and `weights_reg_l1` (bottom) with repeated sampling for categorical hyperparameters, like in Section 4.3.1. Note that the first and the second hyperparameters have respectively a high and low $S_{\mathrm{x}_i,\mathbf{Y}}$. We can see that for hyperparameters with high $S_{\mathrm{x}_i,\mathbf{Y}}$, $\mathrm{u}_i|\mathrm{z} = 1$ (orange for KDE, blue for histogram) is quite different from u_i (red for KDE, gray for histogram). On the contrary, for hyperparameters with low $S_{\mathrm{x}_i,\mathbf{Y}}$ there not seems to have major differences.



(a) MNIST (b) Cifar10 (c) Bateman

Figure 4.9: (top) Histograms of the initial random sampling of configurations and (bottom) comparison of $S_{\mathrm{x}_i,\mathbf{Y}}$ for every main hyperparameters.



(a) MNIST (b) Cifar10 (c) Bateman

Figure 4.10: Representation of $\mathrm{u}_i|\mathrm{z}=1$ (orange for KDE and blue for histogram) and u_i (red for KDE and grey for histogram), for hyperparameters x_i with high (top) and low (bottom) $S_{\mathrm{x}_i,\mathbf{Y}}$

4.4.1.2 Cifar10

We train $n_s = 1000$ different convolutional neural networks. After the initial random search, the best validation error is 81.37%. Note that the histogram of Figure 4.9b is truncated because many hyperparameter configurations led to diverging errors. Here, `pool_type`, `optimizer`, `activation`, `learning_rate` and `kernel_size` have the highest $S_{\mathrm{x}_i,\mathbf{Y}}$, followed by `n_filters`. Half of these hyperparameters are specific to convolutional neural networks, which validates the impact of these layers on classification tasks for image data. The conditional groups are listed in **Appendix B**. We do not show $S_{\mathrm{x}_i,\mathbf{Y}}$ comparisons for every group for clarity of the article and simply report that one conditional hyperparameter `centered`, which triggers centered RMSprop if this value is chosen for `optimizer`, is also found to be impactful.

The depth (`n_layers`) is the less important hyperparameters. Here, the random search yields a neural network of depth 4 and width 53, with 3 stages (meaning that the neural network is widened 3 times), which obtained 80.70% validation accuracy, while the best networks obtained 81.37% accuracy for a depth of 6 and 48 but 4 stages. The conclusion is the same as for MNIST: increasing the size of the network is not the only efficient way to improve its accuracy.

We plot histograms of u_i and $\mathrm{u}_i|\mathrm{z} = 1$ on Figure 4.10b for `pool_type` (top) and `n_layers` (bottom) like in the previous section. The histograms of `n_layers` are interesting because even the histogram of u_i does not seem uniform. An explanation could be that configurations lead to out-of-memory errors or are so long to train that 1000 other neural networks with different configurations have already been trained meanwhile. It also explains why its HSIC is so low. Still, the conclusions that `n_layers` has a limited impact is valid since there is no major differences between u_i and $\mathrm{u}_i|\mathrm{z} = 1$.

4.4.1.3 Bateman equations

For Bateman equations, mean squared error goes down to 2.90×10^{-5}. Like for Cifar10, the histogram of Figure 4.9b is truncated because many hyperparameter configurations led to diverging errors. For this problem, `learning_rate`, `optimizer`, `activations` and `n_layer` can be considered as impactful. Conditional groups are also listed in **Appendix B**. Three conditional hyperparameters are important: `beta_2`, the second moment decay coefficient of Adam and Nadam, `nesterov`, that triggers Nesterov's momentum in SGD and `centered`, described previously.

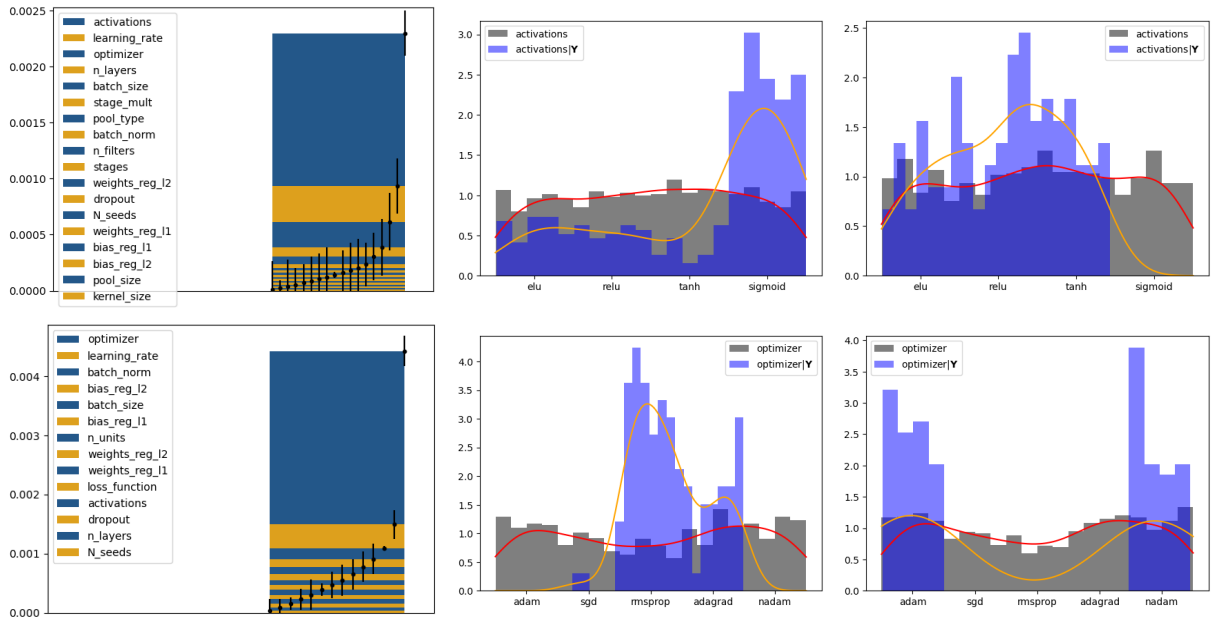
HSIC for `n_layers` is still the lowest of the significant $S_{\mathrm{x}_i,\mathbf{Y}}$ and `n_units` belongs to less impactful hyperparameters. We perform the same analysis as for MNIST and Cifar10 and quote that the best neural network has depth 5 and width 470 while another neural network of depth 5 and width 62 reaches 3.74×10^{-5} validation error.

We plot histograms of u_i and $\mathrm{u}_i|\mathrm{z} = 1$ on Figure 4.10c for `learning_rate` (top) and `bias_reg_l1` (bottom). Histograms of `learning_rate` is interesting because this hyper-

parameter is continuous so the distribution $u_i|z = 1$ seems more natural. This once again illustrates the differences of u_i and $u_i|z = 1$ for hyperparameters with high and low $S_{x_i,\mathbf{Y}}$.

4.4.2 Modification of hyperparameters distribution to improve training stability

Up to now, we only considered $\mathbf{Y}$ to be the 10% best error percentile, which is natural since we want to understand the impact of hyperparameters towards good errors. However, HSIC formalism and our adaptation to hyperparameter analysis allow us to choose any $\mathbf{Y}$. In the previous section, for Cifar10 and Bateman, we truncated histograms of Figure 4.9b because many hyperparameter configurations led to diverging errors. It is possible to understand why by choosing $\mathbf{Y}$ as the set of the 10% worst errors. Then, HSIC can be applied to assess the importance of each hyperparameter towards the worst errors.



(a) $S_{x_i,\mathbf{Y}}$, $\mathbf{Y}$ = 10% worst errors (b) $x_i|\mathbf{Y}$, $\mathbf{Y}$ = 10% worst errors (c) $x_i|\mathbf{Y}$, $\mathbf{Y}$ = 10% best errors

Figure 4.11: **Top**: Cifar10. **Bottom:** Bateman. **(a)** Comparison of $S_{x_i,\mathbf{Y}}$ when $\mathbf{Y}$ is the set of the 10% worst errors. **(b)** Histogram of $x_i|\mathbf{Y}$ when $\mathbf{Y}$ is the set of 10% worst errors, with x_i = `activations` for Cifar10 and x_i = `optimizer` for Bateman. **(c)** Histogram of $x_i|\mathbf{Y}$ when $\mathbf{Y}$ is the set of the 10% best errors, with x_i = `activations` for Cifar10 and x_i = `optimizer` for Bateman.

Figure 4.11b shows $S_{x_i,\mathbf{Y}}$ comparisons, for Cifar10 and Bateman, when $\mathbf{Y}$ is the set of the 10% worst errors. In that case, $S_{x_i,\mathbf{Y}}$ measures how detrimental bad values of x_i can be for the neural network error. For Cifar10, `activation` is the main responsible for the highest errors. If we plot the histogram of `activation`$|\mathbf{Y}$, we can see that `sigmoid` is a bad value in the sense that most of the worst neural networks use this activation function. If we come back to $\mathbf{Y}$ being the set of the 10% best neural networks, we see that none of the best neural

networks have `sigmoid` as the activation function. By itself, this kind of knowledge is valuable because it gives insights about hyperparameter's impact. It also directly brings some practical benefits: in that case, we could reasonably discard `sigmoid` from the hyperparameter space and therefore adapt the distribution of `activation` to improve stability. The same reasoning can be applied to Bateman, with x_i = `optimizer`, for `adagrad` and `rmsprop` optimizers.

Note that we could have drawn the previous conclusions by directly looking at histograms as represented in Figure 4.11b and 4.11c. However, when the number of hyperparameters grows, the number of histograms to look at and visually evaluate grows as well, and the analysis becomes tedious. Thanks to HSIC, we know directly which histograms to look at and how to rank hyperparameters when it is not visually clear-cut.

4.4.3 Interval reduction for continuous or integer hyperparameters that affect execution speed

One common conclusion of $S_{\mathrm{x}_i,\mathbf{Y}}$ values for the last three machine learning problems is that one does not have to set high values for hyperparameters that affect execution speed, such as `n_units`, `n_layers`, or `n_filters`, in order to obtain competitive models. It naturally raises the question of how to bias the hyperparameter optimization towards such models. Multi-objective hyperparameter optimization algorithms have already been successfully applied, like in Tan et al. (2018) for instance, but these algorithms are black-boxes and involve tuning additional hyperparameters for the multi-objective loss function.

In our case, we can use information from $S_{\mathrm{x}_i,\mathbf{Y}}$ to reduce the hyperparameters space search in order to obtain more cost-effective neural networks. The most simple way to achieve that goal is to select values that improve execution speed for hyperparameters which have low $S_{\mathrm{x}_i,\mathbf{Y}}$ values. For MNIST, it would mean for instance to choose `n_units` $= 128$, for Cifar10, `n_layers` $= 3$ or for Bateman, `n_units` $= 32$.

However, if all hyperparameters that affect execution speed are important, i.e. they have high $S_{\mathrm{x}_i,\mathbf{Y}}$ value, we may not be able to apply the previous idea. In that case, we can use HSIC in a different way to still achieve our goal, for integer or continuous hyperparameters (such as `n_layers`, `n_units`, or `kernel_size`). Note that most of the time, for these hyperparameters, a too low or high value will increase the error or the execution speed, respectively. We would like to choose a value which is as low as possible without hurting the error too much. Suppose that x_i = `n_layers` $\in \{a, ..., b\}$ and that $S_{\mathrm{x}_i,\mathbf{Y}}$ is high, so that `n_layers` is among the most important hyperparameters. It is likely that $S_{\mathrm{x}_i,\mathbf{Y}}$ is high because a is too small. One could therefore compute $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,...,b\},\mathbf{Y}}$ for $c \in \{1, ..., b-a\}$, starting with $c = 1$ until $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,...,b\},\mathbf{Y}}$ becomes low. Then, hyperparameter `n_layers` can be replaced by `n_layers`$|$`n_layers` $\in \{a+c, ..., b\}$, which has a low HSIC, and whose value can hence be set to $a + c$.

To illustrate this, let us come back to Runge data set. We first focus on this example because we have been able to train $n_s = 10000$ different neural networks so the methodology can be

tested with limited noise. In Figure 4.12, $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\dots,b\},\mathbf{Y}}$ is plotted with respect to c, where $\mathrm{x}_i = \texttt{n\_layers}$. We see that $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\dots,b\},\mathbf{Y}}$ decreases until $\texttt{n\_layers} = 3$, after which the tendency is not statistically significant. Choosing $\texttt{n\_layers} = 3$ makes `n_layers` belong to the less important hyperparameters so it is a good trade-off value for execution speed and accuracy.

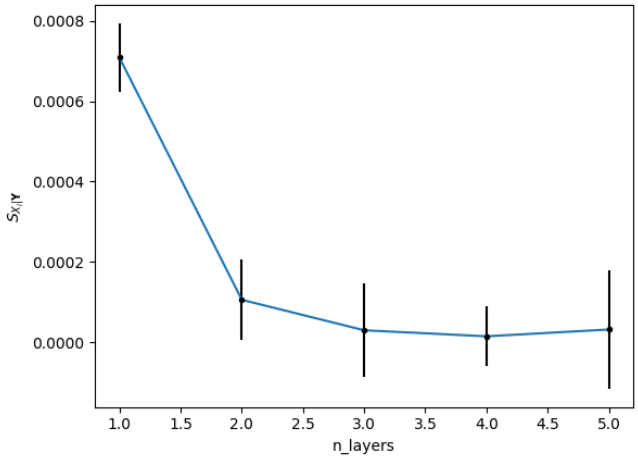
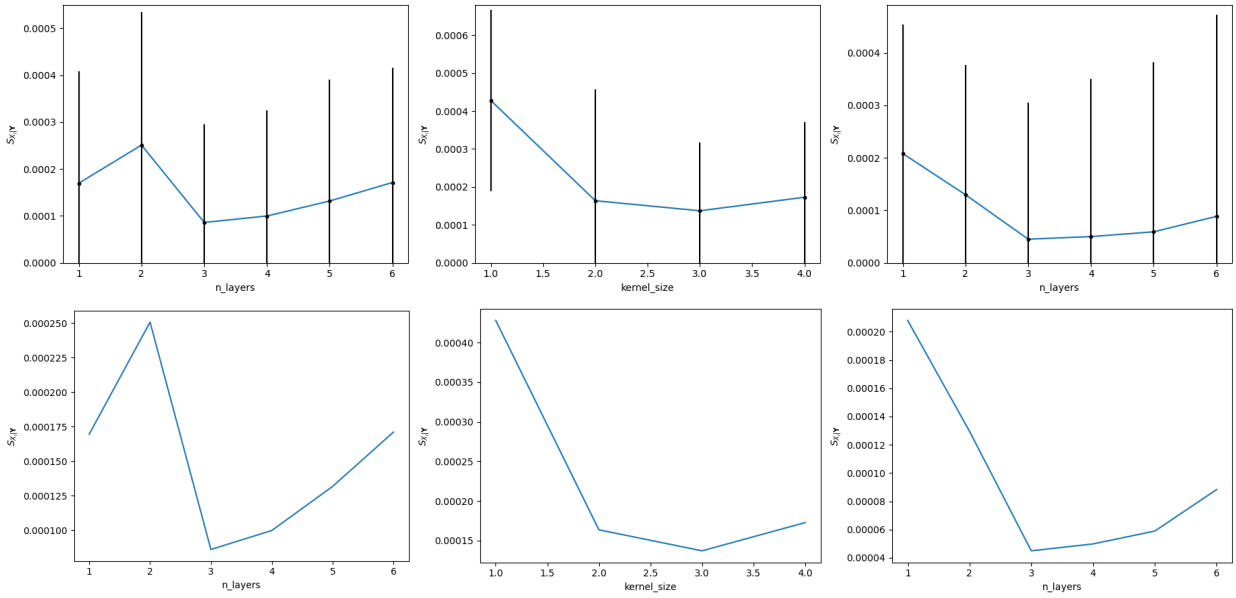


Figure 4.12: $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\dots,b\},\mathbf{Y}}$ w.r.t. c for `n_layers` in Runge. The error bars traduce the standard estimation error.

We apply this methodology to MNIST, Cifar10, and Bateman problems in Figure 4.13. When plotting these curves, too high values of c have to be discarded since the more c increases, the less points there are to compute $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\dots,b\},\mathbf{Y}}$. It could explain the strange behavior of the plots at the right of the axis of Figure 4.13, and the widening of error bars for $\texttt{n\_layers} = 5$ in Figure 4.12.

Note that in Figure 4.13, error bars are much larger than in Figure 4.12. Indeed, in these cases, $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\dots,b\},\mathbf{Y}}$ are evaluated with 10 times less points. Hence, one must be careful with their interpretation. First, for so few estimation points, we are far from the asymptotical regime under which estimation error is gaussian. It explains why error bars can go below 0, whereas the value to estimate is a distance. Therefore, these bars only indicate how spread the error is. Second, since it turns out that the error is very spread, the trade-off value must be chosen with caution by taking this statistic into account. In this manuscript, we rely on a human eye to qualitatively chose this value, but in future work, we should study the use of statistical tests.

Finally, these plots suggests that we could set $\texttt{n\_layers} = 3$ for MNIST, $\texttt{kernel\_size} = 3$ for Cifar10 and $\texttt{n\_layers} = 3$ for Bateman without affecting the error too much. Once the hyperparameters space has been reduced to improve neural networks execution time, it is possible to apply any classical hyperparameter optimization algorithm.



(a) MNIST (b) Cifar10 (c) Bateman

Figure 4.13: $S_{\mathrm{x}_i|\mathrm{x}_i\in\{a+c,\ldots,b\},\mathbf{Y}}$ w.r.t. c for (a) `n_layers` in MNIST, (b) `kernel_size` in Cifar10 and (c) `n_layers` in Bateman with error bars (**top**) and without error bars (**bottom**). The error bars traduce the standard estimation error.

4.5 Optimization by focusing on impactful hyperparameters

One of the most successful and widely used hyperparameter optimization algorithms is Gaussian Processes-based Bayesian Optimization, which we denote GPBO by convenience. However, this algorithm is known to struggle in too high dimensions. In the case of Cifar10, choosing values for hyperparameters that affect execution time would still lead to a space of dimension 20, which is quite large to apply GPBO.

In Song et al. (2007), the authors introduce the use of HSIC for feature selection and in Spagnol et al. (2018), HSIC based feature selection is used in the context of optimization. The idea is to compute $S_{\mathrm{x}_i,\mathbf{Y}}$ for each variable involved in the optimization and to discard low $S_{\mathrm{x}_i,\mathbf{Y}}$ variables from it. More specifically, we fix the discarded variables to an arbitrary value, and then the optimization algorithm is applied only in the dimension of the high $S_{\mathrm{x}_i,\mathbf{Y}}$ variables.

This methodology is particularly suited to hyperparameter optimization. In this work, we have emphasized the ability of HSIC to identify the most important hyperparameters. It allows performing relevant HSIC driven hyperparameters selection, which can overcome optimization in too high dimensional hyperparameters spaces. We go further and present a two-step optimization. In the first step, we optimize the most relevant hyperparameters but in a second step, we also fine-tune less important hyperparameters. As a result, the problematic

optimization in high dimension is split into two easier optimization steps:

1 Optimization in the reduced yet impactful hyperparameters space, which has reasonable dimension. It allows applying GPBO despite the initially large dimension of the hyperparameters space. At the end of this step, optimal values are selected for the most impactful hyperparameters.

2 Optimization on the remaining dimensions. In our case, GPBO can be reasonably applied in this space, but note that we might have hyperparameters spaces whose initial dimension is so high that after the first step, the remaining dimensions to optimize could still be too numerous to perform GPBO. In that case, less refined but more robust hyperparameter optimization algorithms (like random search or Tree Parzen Estimators (Bergstra et al., 2011)) could be applied, which would not be so much of a problem since remaining hyperparameters are less impactful.

For the first step, values have to be chosen for less impactful hyperparameters that are not involved in the optimization. In Spagnol et al. (2018), the authors choose the values yielding the best output after the initial random search. Here, the value selection method that aims at improving execution speed, introduced in Section 4.4.3, integrates perfectly with this two-step optimization. Following this method brings two advantages. First, we can obtain more cost-effective neural networks if we keep these values through the two optimization step. Second, if we do not care so much about execution speed but only look for accuracy, choosing cost-effective values during the first optimization step improves the training speed and thereby global hyperparameter optimization time.

The rest of the low $S_{\mathrm{x}_i,\mathbf{Y}}$ hyperparameters value can be set as those of the hyperparameter configuration yielding the best error. There is one last attention point: one has to be careful about interactions between low $S_{\mathrm{x}_i,\mathbf{Y}}$ hyperparameters. If two low HSIC hyperparameters x_i and x_j are found to interact, like discussed in section 4.3.2, and x_i has an impact on execution speed, the value of x_j must be chosen so that value of the pair $(\mathrm{x}_i, \mathrm{x}_j)$ is close to the value of the hyperparameter configuration of a low error neural network. The two-step optimization is summarized in Algorithm 5.

We evaluate this two-step optimization on our three data sets. For each of these, we consider 4 baselines. For each of these baselines, we report the *test* error (the metric is accuracy for MNIST and Cifar10 and MSE for Bateman), the number of parameters of the best models, and their FLOPs.

- Random search: The result of the random search of 1000 configurations plus 200 additional configurations for a total of $n_s = 1200$ points.
- Full GPBO: Gaussian Processes-based Bayesian Optimization, conducted on the full hyperparameters space, without any analysis based on HSIC. We initialize the optimization with 50 random configurations and perform the optimization for 50 iterations (enough to reach convergence).

Algorithm 5 Two-step Optimization

Inputs: hyperparameter search space $\mathcal{H} = \mathcal{X}_1 \times ... \times \mathcal{X}_{n_h}$, n_s
Apply Algorithm 4: "Evaluation of HSIC in hyperparameter analysis".
Perform interval reduction (for cost efficiency and stability), as in Sections 4.4.3 and 4.4.2.

Select values for less impactful hyperparameters that improve execution speed, taking care of interaction, like discussed in Section 4.3.2.
// `Step 1:`
Apply GPBO to the most impactful hyperparameters.
// `Step 2:`
if goal = accuracy and execution speed **then**
    Keep the optimal values of step 1 and the values of less impactful hyperparameters that improve execution speed. Apply GPBO to the remaining dimensions.
else if goal = accuracy only **then**
    Keep the optimal values of step 1. Apply GPBO to the remaining dimensions.

- TS-GPBO (accuracy): Two-Step GPBO described in Algorithm 5, with goal = accuracy. HSICs are estimated using a first random search of $n_s = 1000$ points. Steps 1 and 2 are run for 25 iterations.
- TS-GPBO (accuracy + speed): Two-Step GPBO described in Algorithm 5, with goal = accuracy and execution speed.

Random search (ran using 100 parallel jobs for MNIST and Bateman and 24 for Cifar10) took between 2 and 3 days depending on the data set, full GPBO between 3 and 4 days and TS-GPBO between 3 and 4 days as well ($2-3$ days for the initial random search and 1 day for the two steps of GPBO). Time measure is coarse because not all the training has been conducted on the same architectures (Sandy Bridge CPUs, Nvidia Tesla V100, and Nvidia Tesla P100 GPUs), even within the same baseline, for cluster accessibility reasons.

We chose the number of total model evaluations for each baseline to obtain approximately the same total execution time. The differences between the number of evaluations, despite identical total execution time, can be explained by different factors. First, the random search can be fully executed in parallel, while GPBO is sequential. Second, step 1 of TS-GPBO always chooses values for non-optimized hyperparameters that improve execution speed and training time. As a result, step 1 is quite fast. Besides, experiments show that step 2 usually converges faster, in terms of the number of evaluations, than full GPBO to the reported minimum, perhaps because the optimal values found during step 1 make step 2 begin close to an optimum. The results of 5 repetitions (except for random search) of each baseline can be found in Table 4.4.

Results show that except for Cifar10, TS+GPBO yields very competitive neural networks while having far fewer parameters and FLOPs. For MNIST, TS+GPBO model has ≈ 66 and 41 times fewer parameters and FLOPs than full GPBO and random search. For Bateman, these

data set	baseline	test metric	params	MFLOPs
MNIST	RS	98.82	6,267,103	12,709 (×41)
-	full GPBO	**98.42** ± 0.05	10,271,367	20,534 (×67)
-	TS-GPBO (accuracy + speed)	**98.42** ± 0.02	**151,306**	**307** (×1)
Cifar10	RS	81.8	99,444,880	1,832,615 (×11)
-	full GPBO	**82.73** ± 1.45	71,111,761	1,441,230 (×8)
-	TS-GPBO (accuracy)	**82.60** ± 0.58	9,604,539	650,269 (×4)
-	TS-GPBO (accuracy + speed)	79.34 ± 0.15	**9,281,258**	**178,621** (×1)
Bateman	RS	**1.99** $\times 10^{-4}$	1,259,140	2,516 (×359)
-	full GPBO	2.94 ± 0.42 $\times 10^{-4}$	1,588,215	3,173 (×453)
-	TS-GPBO (accuracy + speed)	3.49 ± 0.31 $\times 10^{-4}$	**3,291**	**7** (×1)

Table 4.4: Results of hyperparameter optimization for Random Search (RS), Gaussian Processes based Bayesian Optimization on full hyperparameters space (full GPBO) and Two-Steps Gaussian Processes based Bayesian Optimization (TS-GPBO). The mean ± standard deviation across 5 repetitions is displayed for the test metric. For the number of parameters and FLOPs, the maximum value obtained across repetitions is reported because it illustrates the worst scenario that can happen for execution speed and how much our method prevents it.

factors are 482 and 380. An oversized initial hyperparameter search space could explain such a high factor. Still, a reasonable size for the search space cannot be found *a priori*. Note that for these cases, we only reported results of TS-GPBO (accuracy + speed) because the results of this baseline were already satisfying, and TS-GPBO (accuracy) did not bring significant improvement. For the particular case of Cifar10, TS-GPBO (accuracy) and (accuracy + speed) both find a model which has 11 and 9 times fewer parameters than random search and full GPBO. TS-GPBO (accuracy) finds a model with ≈ 3 and 2 fewer FLOPs than random search and full GPBO while these factors are 10 and 8 for (accuracy + speed). Full GPBO and TS-GPBO (accuracy) achieve comparable accuracy, but the standard deviation for full GPBO is 2.5 times higher than for TS-GPBO (accuracy), which demonstrates the robustness of TS-GPBO (accuracy). Even if execution time is not an explicitly desired output of TS-GPBO (accuracy), the first step of TS-GPBO, which selects values that improve execution time, seems to bias the optimization towards more cost-effective models, as the final number of parameters and FLOPs shows. All these results have been allowed thanks to information given by HSIC analysis. Hence, TS-GPBO outputs competitive and cost-effective models but also grants a better knowledge of hyperparameters interaction in these machine learning problems, as opposed to random search and full GPBO.

4.6 Discussion

In this chapter, we have investigated the problem of hyperparameter optimization. Indeed, as we mentioned earlier, this step of supervised deep learning carries high stakes for numerical simulation because hyperparameters can have a high impact on both the accuracy and the cost efficiency of neural networks.

We used sensitivity analysis, an approach that is widely used in numerical and uncertainty analysis. Finally, we designed a methodology that tackles the performance-accuracy trade-off. Furthermore, as the experiments show, the obtained methodology is usable for a broad range of machine learning applications that involve hyperparameters. As a result, in the process of tuning the neural network hyperparameters to conduct supervised deep learning, we used a methodology that comes from numerical and uncertainty analysis.

4.6.1 Impact for deep learning

Many techniques have already been introduced to handle hyperparameter optimization, but they often suffer from a lack of interpretability and interactivity. In this work, we tackled these problems by proposing an HSIC based goal-oriented global sensitivity analysis applied to hyperparameter search spaces. We showcased how we can use this information by improving the stability of training instances and the cost efficiency of trained networks. We also introduced an interpretable hyperparameter optimization methodology that yields competitive and cost-effective neural networks based on feature selection.

These findings are of interest to the machine learning community. Though these example methodologies can be taken as contributions by themselves, they should also be understood as demonstrations that HSIC based goal-oriented global sensitivity analysis is interesting and valuable for hyperparameter optimization. In the end, an important outcome of this work was to make an insightful tool, HSIC, available to the community in the context of hyperparameter optimization.

4.6.2 Impact for numerical simulations

The impact on numerical simulations straightforwardly stems from the fact that we tackled the performance-accuracy trade-off of neural networks. Indeed, we obtained lighter networks without significantly affecting the error, which is the ideal goal when accelerating numerical simulations.

Finally, the results on hyperparameters' relative importance presented in this work have one thing in common. Systematically, the optimizer is at least the second most impactful hyperparameter. It motivates us to work on the optimization of neural networks in order to improve the prediction error and legitimates the next chapter.

4.6.3 Other comments

Other points can be made regarding the presented results and the potential follow-up work. They can be grouped into the following topics:

Hyperparameters modeling choice. HSIC is a powerful tool that is widely used for sensitivity analysis as a dependence measure. Its application to hyperparameter optimization required some work, especially regarding the complex structure of hyperparameter space. To achieve this goal, we made some modeling choices, such as applying Φ_{x_i} to map hyperparameter x_i to a uniform random variable. The good results obtained in Section 4.5 validate not only the usage of information given by HSIC for hyperparameter analysis but also this modeling choice.

Automating Two-Step Gaussian Process-based Bayesian Optimization. In this work, we presented methodologies for exploiting HSIC information that involved human intervention. Indeed, someone has to actively decide which hyperparameter deserves to be considered as more or less impactful. Nevertheless, one advantage of HSIC is that it is a scalar metric. One could construct an HSIC based hyperparameter optimization by setting a threshold above which hyperparameters are considered impactful. It would lead to an end-to-end automatic yet interpretable hyperparameter optimization algorithm. Though Spagnol et al. (2018) use the idea of a threshold, its application to hyperparameter optimization has not been studied in this thesis and could be part of future works.

Other dependence measures. In this work, we used HSIC as a dependence measure. Our derivations for its application to hyperparameter analysis still hold for any other dependence measure sharing the same properties as HSIC, though studies of different dependence measures is beyond the scope of this chapter.

Global hyperparameter optimization speed up. We presented some ways of using HSIC in hyperparameter optimization, but this chapter mainly emphasized the possibility of exploiting it in order to find lighter models. We are aware that execution speed is not always a goal for machine learning practitioners. Still, machine learning practitioners are always concerned about training speed. The first step of TS-GPBO (accuracy) demonstrated the possibility to use HSIC to improve training speed without hurting the final accuracy, so even if final execution speed is not a goal, TS-GPBO made it interesting to use HSIC for that purpose. To go further, it would even be possible to apply parallel GPBO like described in Snoek et al. (2012), or to use Hyperband on the initial random search, since HSIC computation only relies on the error of the p-% best neural networks.

Further execution time improvement. One advantage of execution time improvement obtained thanks to HSIC is that it only relies on choices for the conception of the neural network. Therefore, additional improvements could be made by applying other techniques like quantization, weights pruning, or multi-objective hyperparameter optimization.

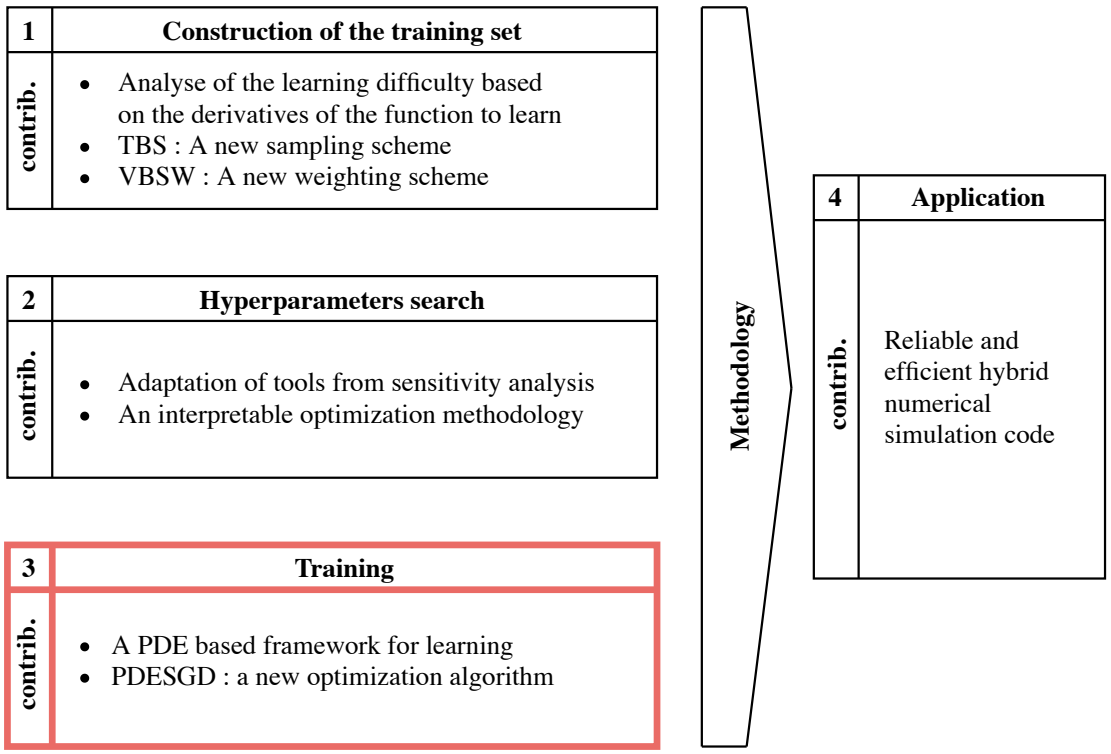
Chapter 5

A view of learning from the Partial Differential Equation theory

The third of the main steps for supervised learning that we identified in Chapter 2 is the learning, or training step. As we mentioned earlier, training a neural network is a non-convex optimization problem. Many techniques are used to tackle this problem, most of which are first and second-order optimization algorithms.

This chapter explores a theoretical question: what is the link between second-order optimization and PDE resolution? To that end, we consider the neural network $f_{\boldsymbol{\theta}}$ as the unknown of a PDE defined on the parameter space Θ, and constructed using the formula of the Newton algorithm. We show that under some conditions, this equation is a drift-diffusion PDE that can be solved by averaging the realizations of a stochastic process. This PDE-based setting provides a learning framework, relying on exploring the parameter space by a stochastic process, that can benefit from PDE resolution theory.

We derive improvements from this framework and enhance classical stochastic gradient descent with new terms that speed up the convergence in convex regions of the parameter space on our experiments. We also use a stability condition from PDE resolution theory that translates into constraints on the learning rate. It allows efficiently exploring the parameter space while maintaining the stability of the process without any learning rate tuning. We showcase the effects of these improvements on a simple two-dimensional optimization problem involving training a neural network with two neurons. A complementary view of the work of the present chapter can be found in Poëtte et al. (2021).



Methodology for supervised deep learning in numerical simulations and contributions of the thesis

Contents

5.1 The optimization task

As it has been mentioned in the previous chapters, the learning of a function $f : \mathbb{R}^{p_{in}} \to \mathbb{R}^{p_{out}}$ by a neural network $f_{\boldsymbol{\theta}}$ consists in minimizing a loss function $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta} \in \Theta$, the parameters of the neural network. Recall that the loss function $J(\boldsymbol{\theta})$ is defined by:

$$J(\boldsymbol{\theta}) = \mathbb{E}[L(f(\boldsymbol{x}, \boldsymbol{\theta}), f(\boldsymbol{x}))] = \int L(f(\boldsymbol{x}, \boldsymbol{\theta}), f(\boldsymbol{x}))d\mathbb{P}_{\mathbf{x}} = \int L(\boldsymbol{x}, \boldsymbol{\theta})d\mathbb{P}_{\mathbf{x}},$$

where $\mathbb{P}_{\mathbf{x}}$ denotes the probability distribution of the data and $L : \mathbb{R}^{p_{out}} \times \mathbb{R}^{p_{out}} \to \mathbb{R}$ denotes a loss function (for instance, the L_2 error). We write $L(\boldsymbol{x}, \boldsymbol{\theta}) = L(f(\boldsymbol{x}), f(\boldsymbol{x}, \boldsymbol{\theta}))$, and $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f(\boldsymbol{x}, \boldsymbol{\theta})$ for conciseness. Hence, learning is an optimization task where the goal is to look for $\boldsymbol{\theta}$ that cancels the gradient

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}},$$

where we note $L' : x, y \to \nabla_y L(x, y)$.

5.1.1 Newton-based algorithms

To cancel the gradient, Newton algorithm uses $\boldsymbol{H}(J)(\boldsymbol{\theta}) = \nabla^2_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to iteratively update $\boldsymbol{\theta}$. The update formula is given by

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \boldsymbol{H}(J)(\boldsymbol{\theta}^k)^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k).$$

This algorithm belongs to a category of methods referred to as second-order optimization methods. Note that most of these methods are derived from the Newton algorithm. However, they are not common in deep learning because the optimization problem is non-convex, and computing the inverse of the Hessian $\boldsymbol{H}(J)(\boldsymbol{\theta}^k)^{-1}$ is a challenging task. Indeed, in a non-convex setting, nothing ensures that $\boldsymbol{H}(J)(\boldsymbol{\theta}^k)^{-1}$ is positive definite hence the optimization may go upward. Moreover, explicit computation of the inverse of the Hessian suffers from a $\mathcal{O}(Card(\boldsymbol{\theta})^3)$ complexity, which is unaffordable for modern neural networks.

The algorithms used in machine learning all alleviate the problem of the Hessian by avoiding its exact computation. Inexact Newton methods compute an approximation of the Hessian,

which is inexact but still brings some computational advantages. For instance, rather than computing $\boldsymbol{H}(J)(\boldsymbol{\theta}^k)^{-1}$, Conjugate Gradient Newton (Golub and Van Loan, 2013) performs a few iterations of the resolution of the system $\boldsymbol{H}(J)(\boldsymbol{\theta}^k)\boldsymbol{s} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k)$ of unknown $\boldsymbol{s}$. Subsampled Newton methods (Byrd et al., 2011) approximate the Hessian with fewer samples, limiting the computational overhead. The default of these algorithms is that they do not naturally deal with non-convexity because they are only designed for convex optimization problems. Some methods are designed to solve this problem (Conn et al., 2000), but none has been established as a gold standard.

Stochastic quasi-Newton methods like L-BFGS (Nocedal and Wright, 1999) or Gauss-Newton methods (Schraudolph, 2002) tackle both problems at the same time by computing an approximation of the Hessian that is ensured to be symmetric positive definite. Finally, some methods only rescale the gradient, which is equivalent to approximating the diagonal of the Hessian. Becker and Cun (1989) compute the diagonal of the Hessian estimated with Gauss-Newton method, while RMSprop (Tieleman et al., 2012) ensures equal optimization progress along each dimension. Empirical improvement on RMSprop, like Adagrad, Adadelta, and Adam (Duchi et al., 2011; Zeiler, 2012; Kingma and Ba, 2015) are very popular in deep learning.

5.1.2 Gradient descent-based algorithms

The most popular algorithms for optimization in deep learning are gradient descent algorithms. The idea is to iteratively update $\boldsymbol{\theta}$ along the direction defined by $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to hopefully reduce $J(\boldsymbol{\theta})$ at each step. The formula for the update of $\boldsymbol{\theta}$ is

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k),$$

where $\gamma > 0$ is a hyperparameter called the learning rate, which can be seen as a step size. This simple algorithm is the basis of a very furnished research area that encompasses all first-order optimization methods for deep learning.

Notice that the gradient descent can be seen as a Newton algorithm that alleviates the problems regarding the computation of $\boldsymbol{H}(J)(\boldsymbol{\theta})$ by assuming that $\boldsymbol{H}(J)(\boldsymbol{\theta})^{-1} = \gamma \boldsymbol{I}_{Card(\boldsymbol{\theta})}$. Besides, RMSprop, Adam, Adadelta, and Adagrad are often considered first-order optimization algorithms because they only involve a rescaling of the gradient, and the rescaling does not explicitly come from an approximation of the Hessian. Stochastic Gradient Descent (SGD, Robbins and Monro (1951)), the stochastic counterpart of gradient descent, is still very popular in its simplest form for optimization in deep learning. Indeed, it is computationally efficient and naturally alleviates the problem of non-convexity because $\gamma > 0$. In addition, its stochastic nature may avoid the process of getting stuck on saddle points in the optimization landscape.

Some methods still aim at improving SGD. They do so by applying momentum to the

optimization trajectory, thereby helping the algorithm escape local minima and accelerating the convergence. Among the most famous algorithms, we can cite heavy ball (Polyak, 1964), which simulates the dynamics of a particle moving in the parameter space and subject to Newton's laws of motion. Another technique is to add Nesterov's momentum (Nesterov, 1983), which ensures the best convergence rate achievable with first-order methods when J is convex and has Lipschitz continuous gradients.

5.1.3 PDE based optimization

Some works intend to cast the optimization at play in machine learning into PDE frameworks. The first inspiration of heavy-ball, as well as the first analysis of its convergence in (Polyak, 1964), was based on second-order ODE analysis. Yang et al. (2018) revisit this approach and introduce a more general framework that encompasses more diverse techniques such as SGD, Newton's methods, and their variants accelerated by Nesterov's momentum. This framework enables a more thorough analysis of these algorithms but only aims at gaining insights on their behavior and is not designed to improve them.

Some works model the successive iterations of SGD as a stochastic process that solves a diffusion PDE. Mobahi (2016) introduces an algorithm that brings many advantages stemming from the analogy with PDE resolution. For instance, the process explicitly adds stochasticity, and the framework dynamically controls the learning rate. Gelfand and Mitter (1991) introduce Langevin SGD, which explicitly adds noise to the SGD to help it escape from local minima. This algorithm was later analyzed by Raginsky et al. (2017), which draws a parallel between Langevin SGD and a continuous-time diffusion process.

However, none of these works exploits the unique structure of the learning problem based on the composition of L and $f_{\boldsymbol{\theta}}$. In our work, we take this specificity into account, similarly to the Levenberg-Marquardt algorithm (Marquardt, 1963) for least-squares problems. In addition, we use our framework to derive a new optimization algorithm, which empirically exhibits original stability and accuracy properties.

5.2 Stochastic resolution of Partial Differential Equations

In this section, we point out the link that can be made between optimization and resolution of PDEs. These derivations assume that $p_{out} = 1$, which still spans a large range of machine learning problems. Then, we remind some tools of PDE resolution theory that are needed to exploit this link.

5.2.1 A link between optimization and resolution of PDE

The update formula of Newton algorithm can be reformulated as :

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \boldsymbol{H}(J)(\boldsymbol{\theta})(\boldsymbol{\theta}^* - \boldsymbol{\theta}) = 0,$$

where $\boldsymbol{\theta}$ is the current weights values and $\boldsymbol{\theta}^*$ is the updated value that we are looking for. This formulation can be written based on f and L :

$$\begin{aligned} 0 = &\int L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}} \\ &+ \int \Big[L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) + L''(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) \nabla^T_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) \Big] (\boldsymbol{\theta}^* - \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}}. \end{aligned} \tag{5.1}$$

equation (5.1) can be seen as a Partial Differential Equation (PDE), of unknowns $\boldsymbol{\theta}^*$ and $\boldsymbol{\theta} \to f(\boldsymbol{x}, \boldsymbol{\theta})$, describing the update process of $\boldsymbol{\theta}$. Note that this PDE system is unclosed because we only have one equation for two unknowns. This formulation still emphasizes the previously described dimension problems stemming from the computation of $\nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta})$. However, it enables us to use tools from numerical analysis to try to overcome this problem. In the following, we focus on Monte Carlo (MC) resolution of PDEs, which is particularly suited to high-dimension problems.

5.2.2 Background on stochastic resolution of PDE

In order to explicitly bridge the gap between learning and stochastic resolution of PDE, we need to introduce some theoretical tools. The formulations and notations are taken from Oksendal (1992); Stroock and Varadhan (1997).

Definition (Ito process)**.** *Let $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to \boldsymbol{\mu}(t, \boldsymbol{x}) \in \mathbb{R}^n$, $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to \boldsymbol{\Sigma}(t, \boldsymbol{x}) \in \mathbb{R}^{n \times m}$ and dB^t be an m dimensional Brownian process. Then, the process X^t defined by*

$$dX^t = \boldsymbol{\mu}(t, X^t)dt + \boldsymbol{\Sigma}(t, X^t)dB^t, \tag{5.2}$$

is an Ito process.

Lemma 2 (Ito Formula)**.** *Assume X^t is an n dimensional Ito process. Let $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to g(t, \boldsymbol{x}) = (g_1(t, \boldsymbol{x}), ..., g_p(t, \boldsymbol{x}))^T \in \mathbb{R}^p$ with g being in $\mathcal{C}^2$. Then $Y^t = g(t, X^t)$ is a p dimensional Ito process and verifies $\forall k \in \{0, ..., p\}$*

$$dY^t_k = \partial_t g_k(t, X^t)dt + \sum_{i=1}^n \partial_{x_i} g_k(t, X^t) dX^t_i + \frac{1}{2} \sum_{i,j=1}^n \partial^2_{x_i, x_j} g_k(t, X^t) dX^t_i dX^t_j.$$

Lemma 3 (Ito lemma)**.** *Let X^t be an Ito process following equation (5.2), with $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to \boldsymbol{\mu}(t, \boldsymbol{x}) \in \mathbb{R}^n$ and $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to \boldsymbol{\Sigma}(t, \boldsymbol{x}) \in \mathbb{R}^{n \times m}$. Let $t, \boldsymbol{x} \in \mathbb{R}^+ \times \mathbb{R}^n \to g(t, \boldsymbol{x}) = (g_1(t, \boldsymbol{x}), ..., g_p(t, \boldsymbol{x}))^T \in \mathbb{R}^p$ with g being in $\mathcal{C}^2$ and $Y^t = g(t, X^t)$. Then we have $\forall k \in \{1, ..., p\}$*

$$\begin{aligned} dY_k^t = &\left[\partial_t g_k(t, X^t)dt + \sum_{i=1}^{n} \partial_{x_i} g_k(t, X^t)\mu_i(t, X^t) + \frac{1}{2}\sum_{i,j=1}^{n} \partial^2_{x_i,x_j} g_k(t, X^t)[\boldsymbol{\Sigma}(t, X^t)\boldsymbol{\Sigma}^T(t, X^t)]_{i,j}\right]dt \\ &+ \sum_{i=1}^{n} \partial_{x_i} g_k(t, X^t) \sum_{j=1}^{m} \boldsymbol{\Sigma}_{i,j}(t, X^t)dB^t, \end{aligned} \tag{5.3}$$

where $[\boldsymbol{\Sigma}(t, X^t)\boldsymbol{\Sigma}^T(t, X^t)]_{i,j} = \sum_{k=1}^{m} \boldsymbol{\Sigma}_{i,k}(t, X^t)\boldsymbol{\Sigma}_{j,k}(t, X^t)$. The previous equation (5.3) can be formulated in a matrix form, for conciseness :

$$\begin{aligned} dY^t = &\left[\partial_t g(t, X^t)dt + \nabla_{\boldsymbol{x}} g(t, X^t)\boldsymbol{\mu}(t, X^t) + \frac{1}{2} Tr[\boldsymbol{\Sigma}(t, X^t)^T \nabla^2_{\boldsymbol{x}} g(t, X^t)\boldsymbol{\Sigma}(t, X^t)]\right]dt \\ &+ \nabla_{\boldsymbol{x}} g(t, X^t)\boldsymbol{\Sigma}(t, X^t)dB_j^t. \end{aligned} \tag{5.4}$$

In the previous definitions and lemma, we have introduced X^t, an Ito process, and $Y^t = g(t, X^t)$. In order to tackle the learning problem defined in equation (5.1), we are going to construct $\boldsymbol{\theta}$ as an Ito process and f as a process constructed on $\boldsymbol{\theta}$, with appropriate assumptions on $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. One last tool is needed to work on bridging the simulation of such a process with our PDE resolution.

Theorem 1 (Kolmogorov backward equation)**.** *Let f^0 be in $\mathcal{C}^2$ and let X^t be an Ito process as in equation (5.2) with initial condition $X^0 = \boldsymbol{s}$. Kolmogorov's theorem states that $f_{\boldsymbol{s}}(t, \boldsymbol{s})$ defined as $f_{\boldsymbol{s}}(t, \boldsymbol{s}) = \mathbb{E}_{X^t}[f^0(\boldsymbol{x})]$ is solution of*

$$\begin{cases} \partial_t f_{\boldsymbol{s}}(t, \boldsymbol{s}) + \sum_{i=1}^{n} \partial_{s_i} f_{\boldsymbol{s}}(t, \boldsymbol{s})\mu_i(t, \boldsymbol{s}) + \frac{1}{2}\sum_{i,j=1}^{n} \partial^2_{s_i,s_j} f_{\boldsymbol{s}}(t, \boldsymbol{s})[\boldsymbol{\Sigma}(t, \boldsymbol{s})\boldsymbol{\Sigma}^T(t, \boldsymbol{s})]_{i,j} = 0, \\ f_{\boldsymbol{s}}(0, \boldsymbol{s}) = f^0(\boldsymbol{s}). \end{cases} \tag{5.5}$$

The terms $\boldsymbol{\mu}(t, \boldsymbol{s})$ and $[\boldsymbol{\Sigma}(t, \boldsymbol{s})\boldsymbol{\Sigma}^T(t, \boldsymbol{s})]$ are the drift term and diffusion coefficients. This theorem states that we can solve a drift-diffusion equation by averaging realizations of a stochastic process, $f^0(X^t)$, which we can simulate based on an Ito process, X_t, thanks to Ito's lemma. At this point, we have introduced all the mathematical tools needed to bridge the gap between the learning task and stochastic resolution of PDEs. Yet, we are still far from this goal. The next section establishes this link.

5.3 Learning task formulated as a stochastic PDE

Using the tools previously described in the context of learning requires linking equation (5.1) to equation (5.5) - as of now, those two equations may still seem very different. Indeed, the former is not formulated as a drift-diffusion equation, is stationary, and still depends on $\mathbf{x}$, unlike the latter. In a first section, we formulate equation (5.1) as a drift-diffusion PDE and close the gap with equation (5.5). Then, we introduce discretization methods to solve this equation.

5.3.1 The optimization step as a drift-diffusion equation

In this section, we emphasize the link between the PDE formulation of the optimization step introduced in equation (5.1) and a drift-diffusion PDE that can be solved stochastically using Ito processes and Kolmogorov's equation. To that end, we have to get equation (5.1) as close as possible to equation (5.5).

There is a first link which can be established immediately. The variable $\boldsymbol{s}$ echoes $\boldsymbol{\theta}$ so the function $f_{\boldsymbol{s}}$ echoes f and since they live in $\mathbb{R}^{p_{out}}$ and $\mathbb{R}^p$, p echoes p_{out}. Moreover, n echoes $Card(\boldsymbol{\theta})$. In the following, we denote p_{out} by p and $Card(\boldsymbol{\theta})$ by n.

The second link is less straightforward but can still be established quite easily. It alleviates the fact that equation (5.1) is vectorial while equation (5.5) is not. Let us introduce an arbitrary vector $\boldsymbol{\alpha} \in \mathbb{R}^n$. Then, equation (5.1) is equivalent to

$$\begin{aligned} 0 = &\int L'(\boldsymbol{x},\boldsymbol{\theta})\boldsymbol{\alpha}^T \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}} \\ &+ \int \boldsymbol{\alpha}^T \Big[L'(\boldsymbol{x},\boldsymbol{\theta}) \nabla^2_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}) + L''(\boldsymbol{x},\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}) \nabla^T_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}) \Big] (\boldsymbol{\theta}^* - \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}}, \\ &\forall \boldsymbol{\alpha} \in \mathbb{R}^n. \end{aligned} \tag{5.6}$$

This equivalency is ensured only if equation (5.6) is solved $\forall \boldsymbol{\alpha} \in \mathbb{R}^n$ or for $\boldsymbol{\alpha}$ being the components of a basis of $\mathbb{R}^n$. To make it even more look like equation (5.5), we can reformulate equation (5.6) to group the derivatives of f of the same order. Then, we have

$$\begin{aligned} 0 = &\sum_{i=1}^{n} \int \partial_{\theta_i} f_k(\boldsymbol{x},\boldsymbol{\theta}^t) \alpha_i \left[L'(\boldsymbol{x},\boldsymbol{\theta}^t) + L''(\boldsymbol{x},\boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f_k(\boldsymbol{x},\boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] d\mathbb{P}_{\mathbf{x}} \\ &+ \frac{1}{2} \sum_{i,j=1}^{n} \int \partial^2_{\theta_i,\theta_j} f_k(\boldsymbol{x},\boldsymbol{\theta}^t) 2\alpha_i L'(\boldsymbol{x},\boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}}, \\ &\forall \boldsymbol{\alpha} \in \mathbb{R}^n, k \in \{1, ..., p\}. \end{aligned}$$

The other problems we still have to deal with are more difficult and deserve their own section. These problems are :

- Equation (5.5) is unstationary, i.e. there is a dependance in time and an additional time dependent term, $\partial_t f_{\boldsymbol{s}}(t, \boldsymbol{s})$, whereas equation (5.6) is stationary.
- Equation (5.6) is integrated on $d\mathbb{P}_{\mathbf{x}}$, while $\mathbf{x}$ is not explicitly found in equation (5.5).

To focus on these problems, from now on, we assume for clarity and without loss of generality that $p_{out} = p = 1$. Hence, f_k becomes f in the following equations.

5.3.1.1 From stationary to unstationary processes

As we just saw, in equation (5.6), $f_{\boldsymbol{s}}$ echoes f but $f_{\boldsymbol{s}}$ depends on t, unlike f. Let us consider the following unstationary equation

$$\begin{aligned} 0 =& \partial_t \int f(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}} \\ &+ \sum_{i=1}^{n} \int \partial_{\theta_i} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \alpha_i \left[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) + L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] d\mathbb{P}_{\mathbf{x}} \\ &+ \frac{1}{2} \sum_{i,j=1}^{n} \int \partial^2_{\theta_i, \theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t) 2\alpha_i L'(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}}, \\ &\forall \boldsymbol{\alpha} \in \mathbb{R}^n, k \in \{1, ..., p\}. \end{aligned} \tag{5.7}$$

This equation must be solved $\forall t \in [0, T]$ and $\forall \boldsymbol{\theta} \in \Theta$. We consider the case with $T < \infty$ and $|\Theta| \leq \infty$, where equation (5.7) together with an initial condition $f^0(\boldsymbol{\theta}) = f(\boldsymbol{x}, \boldsymbol{\theta}^0)$ is a Cauchy problem. In that case, the problem is well posed (Brezis, 2010).

However, introducing nonstationarity implies changing the initial approach. Indeed, equation (5.1) describes one step of a Newton algorithm, so solving this PDE returns $\boldsymbol{\theta}^*$, the next update value for $\boldsymbol{\theta}$. Here, we simulate a nonstationary process that satisfies equation (5.7). Thus, $\boldsymbol{\theta}^* - \boldsymbol{\theta}$ can be seen as a parameter shift. We still have to model this shift, which will be discussed later, but it is important to note that the simulated process now explores the parameter space as much as it optimizes $J(\boldsymbol{\theta})$.

Now, since f depends on t through $\boldsymbol{\theta}^t$, $J(\boldsymbol{\theta}^t)$ also does, and the optimal θ_{t^*} is obtained by monitoring $J(\boldsymbol{\theta}^t)$ (or any reference metric, like for instance the validation error) throughout the simulation. In other words, in that case,

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}^{t^*} \quad \text{with} \quad t^* = \underset{t \in [0,T]}{\operatorname{argmin}} J(\boldsymbol{\theta}^t).$$

Remark. *This methodology is similar to many simulation codes, especially in computational physics (e.g. Kluth and Després (2010); Maire et al. (2007a)). We are often interested in a metric based upon the simulated phenomenon. For instance, in solid mechanics, we may monitor the main stress of material under simulated constraints. Recording the maximum stress (which would be analogous to $J(\boldsymbol{\theta}^{t^*})$) allows assessing if the material would break or not under these constraints.*

5.3.1.2 The dependence with respect to x

First of all, note that solving equation (5.7) is equivalent to solving $\forall \mathbf{x}, \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}$,

$$\begin{cases} 0 = \partial_t f(\mathbf{x}, \boldsymbol{\theta}^t) \\ \quad + \sum_{i=1}^{n} \partial_{\theta_i} f(\mathbf{x}, \boldsymbol{\theta}^t) \alpha_i \left[L'(\mathbf{x}, \boldsymbol{\theta}^t) + L''(\mathbf{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\mathbf{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] \\ \quad + \frac{1}{2} \sum_{i,j=1}^{n} \partial^2_{\theta_i, \theta_j} f(\mathbf{x}, \boldsymbol{\theta}^t) 2\alpha_i L'(\mathbf{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t). \\ f(\mathbf{x}, \boldsymbol{\theta}^0) = f^0(\boldsymbol{\theta}), \quad \boldsymbol{\theta} \in \Theta, \quad t \in [0, T], \quad \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}. \\ \text{together with the evaluation of the error metric.} \end{cases} \tag{5.8}$$

This equation is very similar to equation (5.7). The only difference is that it has to be solved $\forall \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}$. Equation (5.8) is a typical formulation of an uncertainty quantification problem, and many methods related to this research field are able to handle it, see Carrillo and Zanella (2019); Pareschi (2020); Poëtte (2020, 2019a,b); Le Maitre et al. (2002). However, to stick with the context of learning, we have to go further.

Let us decompose $f(\boldsymbol{x}, \boldsymbol{\theta}^t)$ into

$$f(\boldsymbol{x}, \boldsymbol{\theta}^t) = \bar{f}(\boldsymbol{\theta}^t) + \hat{f}(\boldsymbol{x}, \boldsymbol{\theta}^t),$$

where

$$\bar{f}(\boldsymbol{\theta}^t) = \int f(\boldsymbol{x}, \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}} \text{ and } \hat{f}(\boldsymbol{x}, \boldsymbol{\theta}^t) = f(\boldsymbol{x}, \boldsymbol{\theta}^t) - \bar{f}(\boldsymbol{\theta}^t).$$

The function $\bar{f}$ is the mean of f over $d\mathbb{P}_{\mathbf{x}}$ and $\hat{f}$ is a centered fluctuation. This kind of decomposition is intensively used in turbulence modeling, see Majda (1984). We then have to solve

$$
\begin{cases}
0 = \ \partial_t \bar{f}(\mathbf{x}, \boldsymbol{\theta}^t) \\
\quad + \sum_{i=1}^{n} \partial_{\theta_i} \bar{f}(\mathbf{x}, \boldsymbol{\theta}^t) \int \alpha_i \left[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) + L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] d\mathbb{P}_{\mathbf{x}} \\
\quad + \frac{1}{2} \sum_{i,j=1}^{n} \partial^2_{\theta_i, \theta_j} \bar{f}(\mathbf{x}, \boldsymbol{\theta}^t) 2 \int \alpha_i L'(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}} \\
\quad + \sum_{i=1}^{n} \int \partial_{\theta_i} \hat{f}(\boldsymbol{x}, \boldsymbol{\theta}^t) \alpha_i \left[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) + L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] d\mathbb{P}_{\mathbf{x}} \\
\quad + \frac{1}{2} \sum_{i,j=1}^{n} \int \partial^2_{\theta_i, \theta_j} \hat{f}(\boldsymbol{x}, \boldsymbol{\theta}^t) 2 \alpha_i L'(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}}, \\
f(\boldsymbol{x}, \boldsymbol{\theta}^0) = f^0(\boldsymbol{\theta}), \quad \boldsymbol{\theta} \in \Theta, \quad t \in [0, T], \quad \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}.
\end{cases}
\tag{5.9}
$$

The above PDE is not closed in the sense that we have one equation but two unknowns $\bar{f}$ and $\hat{f}$. One way to solve it is to choose a heuristic to add a second equation to the system. In order to close the gap between equation (5.9) and equation (5.5), the most straightforward hypothesis to make is that $\hat{f}$ is negligible in the resolution of equation (5.9). It adds a second equation which is $\hat{f} \equiv 0$, and yields the following PDE system :

$$
\begin{cases}
0 = \ \partial_t \bar{f}(\boldsymbol{\theta}^t) \\
\quad + \sum_{i=1}^{n} \partial_{\theta_i} \bar{f}(\boldsymbol{\theta}^t) \int \alpha_i \left[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) + L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \right] d\mathbb{P}_{\mathbf{x}} \\
\quad + \frac{1}{2} \sum_{i,j=1}^{n} \partial^2_{\theta_i, \theta_j} \bar{f}(\boldsymbol{\theta}^t) 2 \alpha_i \int L'(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}}, \\
f(\boldsymbol{x}, \boldsymbol{\theta}^0) = f^0(\boldsymbol{\theta}), \quad \boldsymbol{\theta} \in \Theta, \quad t \in [0, T], \quad \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}.
\end{cases}
\tag{5.10}
$$

Then, it is possible to solve equation (5.10) by identifying it with equation (5.5) and simulating the corresponding Itô process.

5.3.2 Learning by simulating a stochastic process

In the previous part, we have constructed a formalization of the learning problem that enables its resolution using equation (5.5), namely the Kolmogorov backward equation. Still, equation (5.10) can not be solved as such. We need to choose $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ to match equation (5.5). This implies to choose values for $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}^t$. Moreover, the definition of $\boldsymbol{\theta}^t$ and of $\mathbb{P}_{\mathbf{x}}$ is continuous, and since equation (5.10) can not be solved analytically, we have to focus on their discretization.

5.3.2.1 Identification of the drift and diffusion coefficients

In equation (5.10), the drift term $\boldsymbol{\mu}$ can be easily identified. We have :

$$\boldsymbol{\mu}_i(\boldsymbol{\theta}^t) = \int \alpha_i \Big(L'(\boldsymbol{x}, \boldsymbol{\theta}^t) + L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \sum_{j=1}^{n} \partial_{\theta_j} f(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) \Big) d\mathbb{P}_{\mathbf{x}}.$$

However, identifying the diffusion coefficients, $\boldsymbol{\Sigma}$ is not so straightforward because it implies to find $\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)$ such that :

$$\big[\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)\boldsymbol{\Sigma}^T(t, \boldsymbol{\theta}^t)\big]_{i,j} = 2 \int \alpha_i L'(\boldsymbol{x}, \boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) d\mathbb{P}_{\mathbf{x}}. \tag{5.11}$$

Lemma 4. *A decomposition as in equation (5.11) always exists.*

Proof. Let us first remark that the matrix $\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)\boldsymbol{\Sigma}^T(t, \boldsymbol{\theta}^t)$ is symmetric by construction. Hence, we must have $\forall i, j \in \mathbb{R}^n$,

$$\big[\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)\boldsymbol{\Sigma}^T(t, \boldsymbol{\theta}^t)\big]_{i,j} = \big[\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)\boldsymbol{\Sigma}^T(t, \boldsymbol{\theta}^t)\big]_{j,i}. \tag{5.12}$$

We can always satisfy this constraint by taking $\forall i, j \in \{1, ..., n\}^2$,

$$\alpha_i(\theta_j^* - \theta_j^t) = \alpha_j(\theta_i^* - \theta_i^t).$$

This property is verified if we choose $\alpha_i = 0$ or $\theta_i^* - \theta_i^t = 0$, $\forall i \in \{1, ..., n\}$ but the former is trivial and the latter removes the stochasticity from the problem, which is not desirable. The only other possibility is that $\boldsymbol{\alpha} \propto \boldsymbol{\theta}^* - \boldsymbol{\theta}^t$. In that case, it is possible to choose

- $\boldsymbol{\theta}^* - \boldsymbol{\theta}^t = \int L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \mathbf{1}_n d\mathbb{P}_{\mathbf{x}}$
- $\boldsymbol{\alpha} = \mathbf{1}_n$,

where $\mathbf{1}_n = (1, ..., 1)^T \in \mathbb{R}^n$, which allows defining :

$$\boldsymbol{\Sigma}(\boldsymbol{\theta}^t) = \frac{2}{\sqrt{n}} \left| \int L'(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}} \right| \mathbf{1}_n {\mathbf{1}_n}^T. \tag{5.13}$$

□

Note that there may be other possible decomposition that satisfy equation (5.12) (see Section 5.4.1 for another example). Every possible decomposition choice could be seen as a different heuristic to solve the minimization problem. However, in practice, since $\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)$ is in factor of $\mathbf{g}$, it is not mandatory to perform the decomposition since one can simply directly sample from $\mathcal{N}(0, \boldsymbol{\Sigma}(\boldsymbol{\theta}^t)\boldsymbol{\Sigma}^T(t, \boldsymbol{\theta}^t))$.

In the following, we assume the existence of $\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)$ since we can always at least choose a decomposition like in equation (5.13). Moreover, we denote $\boldsymbol{\Sigma}(\boldsymbol{\theta}^t)$ by $\int \boldsymbol{\Sigma}(\boldsymbol{x}, \boldsymbol{\theta}^t, \boldsymbol{\alpha}, \boldsymbol{\theta}^* - \boldsymbol{\theta}^t) d\mathbb{P}_X$ to account for the possible choices of $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}^t$ and their impact on the integration with respect to $\mathbf{x}$. Indeed, they can be chosen such that they depend on $\boldsymbol{x}$ and $\boldsymbol{\theta}^t$. Nonetheless, we do not include these dependencies in the notations for clarity and because they are not mandatory.

5.3.2.2 Discretization schemes

Equation (5.10) still can not be solved as such because $\boldsymbol{\theta}^t$ is defined as a continuous Itô process, and $\mathbb{P}_{\mathbf{x}}$ is a continuous probability distribution. In practice, equation (5.10) can not be solved analytically, and $\boldsymbol{\theta}^t$ has to be simulated numerically. Moreover, we only have access to a finite set of samples of $\mathbb{P}_{\mathbf{x}}$, which is the training data set.

Discretization of $\boldsymbol{\theta}^t$: Thanks to Kolmogorov's backward equation, we can solve equation (5.10) by averaging the stochastic process

$$\begin{aligned}\boldsymbol{\theta}^t = \boldsymbol{\theta}^0 &+ \int \int_0^t \left[\boldsymbol{\alpha} L'(\boldsymbol{x}, \boldsymbol{\theta}^s) + L''(\boldsymbol{x}, \boldsymbol{\theta}^s)\boldsymbol{\alpha}\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^s)(\boldsymbol{\theta}^* - \boldsymbol{\theta}^s)\right] d\mathbb{P}_{\mathbf{x}} ds \\ &+ \int \int_0^t \boldsymbol{\Sigma}(\boldsymbol{x}, \boldsymbol{\theta}^s, \boldsymbol{\alpha}, \boldsymbol{\theta}^* - \boldsymbol{\theta}^s) d\mathbb{P}_{\mathbf{x}} dB^s,\end{aligned}$$

with B^s a m dimensional Brownian motion. Many discretization schemes can be used in order to simulate this stochastic process. In this work, for simplicity, we consider an explicit Euler discretization scheme over time step $[t^k, t^{k+1} = t^k + \Delta t]$. The process $\boldsymbol{\theta}^t$ can then be simulated using the expression

$$\begin{aligned}\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k &+ \Delta t \int \left[\boldsymbol{\alpha} L'(\boldsymbol{x}, \boldsymbol{\theta}^k) + L''(\boldsymbol{x}, \boldsymbol{\theta}^k)\boldsymbol{\alpha}\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k)(\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right] d\mathbb{P}_{\mathbf{x}} \\ &+ \sqrt{\Delta t}\left[\int \boldsymbol{\Sigma}(\boldsymbol{x}, \boldsymbol{\theta}^k, \boldsymbol{\alpha}, \boldsymbol{\theta}^* - \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}\right]\mathbf{g},\end{aligned} \tag{5.14}$$

with $\mathbf{g} = (\mathrm{g}_1, ..., \mathrm{g}_m)$ and $\mathrm{g}_i \sim \mathcal{N}(0, 1)$ independent and identically distributed.

Remark. *Equation (5.14) implies to choose a discretization time step Δt. This hyperparameter is analogous to the learning rate γ, which can be seen as a time step on the optimization*

trajectory of gradient descent.

Discretization of $\mathbb{P}_{\mathbf{x}}$: The discretization of $\mathbb{P}_{\mathbf{x}}$ is ubiquitous in machine learning since, most of the time, we only have access to $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, a set of samples from this distribution. In such cases, the solution is to use Monte Carlo integration. Hence, equation (5.14) becomes

$$\begin{aligned}\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \frac{\Delta t}{N}\sum_{i=1}^{N}\left[\boldsymbol{\alpha}L'(\boldsymbol{x}_i, \boldsymbol{\theta}^k) + L''(\boldsymbol{x}_i, \boldsymbol{\theta}^k)\boldsymbol{\alpha}\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i, \boldsymbol{\theta}^k)(\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right] \\ + \frac{\sqrt{\Delta t}}{N}\left[\sum_{i=1}^{N}\boldsymbol{\Sigma}(\boldsymbol{x}_i, \boldsymbol{\theta}^k, \boldsymbol{\alpha}, \boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right]\mathbf{g}.\end{aligned} \tag{5.15}$$

Remark. *In equation (5.15), we need to compute $\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i, \boldsymbol{\theta}^k)$ in order to obtain $\boldsymbol{\theta}^{k+1}$. The PDE formulation of the problem does not alleviate the need to use the gradient back-propagation algorithm.*

Now, thanks to equation (5.15), we have a general, PDE-based framework for optimizing $\boldsymbol{\theta}$ in order to minimize any test error based on $f_{\boldsymbol{\theta}}$. All that is left is to simulate $\boldsymbol{\theta}^t$ and assess our test metric of interest at any iteration. For suitable choices of $\boldsymbol{\alpha}$, $\boldsymbol{\theta}^* - \boldsymbol{\theta}$ and in the regime where $\hat{f} \equiv 0$, Kolmogorov backward equation guarantees that this process solves our minimization problem. It naturally raises the question of whether the assumption $\hat{f} \equiv 0$, and the possibility to choose $\boldsymbol{\alpha}$, $\boldsymbol{\theta}^* - \boldsymbol{\theta}$ are reasonable or not. The following sections tend to comfort these assumptions. Indeed, we show that SGD, well known for its empirical success, falls under our PDE framework. Moreover, we design a new and promising optimization algorithm using specific choices of $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}$.

5.4 A PDE-consistent Stochastic Gradient Descent

The PDE-based framework for learning based on the simulation of $\boldsymbol{\theta}^t$ as a stochastic process leaves the possibility to choose $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}^t$. As we mentioned earlier, different choices would possibly lead to other heuristics and different algorithms. That said, we could investigate to what extent already existing optimization algorithms used in Deep Learning fall under this framework. As a first step towards a complete answer to this broad question, we focus on the most commonly used one: Stochastic Gradient Descent (SGD). We first emphasize how SGD is related to our PDE framework. Then, thanks to well-known properties from PDE resolution theory, we introduce a new "PDE-consistent" SGD that we test on a simple example. This algorithm is an illustration of how learning may benefit from numerical analysis.

5.4.1 SGD as a stochastic process simulation

To emphasize the link between SGD and the PDE framework, let us first come back to the definition of Gradient Descent (GD). Recall that, in its more general form, one iteration of GD can be written

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k),$$

with

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \int L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}) d\mathbb{P}_{\mathbf{x}}.$$

The gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is estimated using a training data set $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ of N samples drawn from $\mathbb{P}_{\mathbf{x}}$. Hence, one iteration of GD follows the expression

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\gamma}{N} \sum_{i=1}^{N} L'(\boldsymbol{x}_i, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) = \boldsymbol{\theta}^k - \gamma \int L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N,$$

where $d\mathbb{P}_{\mathbf{x}}^N = \frac{1}{N} \sum_{i=1}^{N} \delta(\boldsymbol{x}_i - \mathbf{x})$ where δ is the Dirac measure. In practice, in deep learning, SGD is almost always used instead of GD. SGD consists in randomly selecting a subset $\{\boldsymbol{x}_{p(1)}, ..., \boldsymbol{x}_{p(N_b)}\}$ of the training data set $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, with $N_b < N$ a hyperparameter called batch size and $g : \{1, ..., N_b\} \to \{1, ..., N\}$ an injective application randomly defined at each batch construction. Then, the update formula is given by

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\gamma}{N_b} \sum_{i=1}^{N_b} L'(\boldsymbol{x}_{g(i)}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_{g(i)}, \boldsymbol{\theta}^k).$$

Remark. *This preference can be justified by different reasons. First, when the training data set is too large, estimating the gradient with the whole data set may be too expensive in terms of hardware memory consumption. Moreover, some empirical results show that with a reduced batch size (Keskar et al., 2017; Izmailov et al., 2018a), the obtained neural networks generalize better, although it is still very discussed (Hoffer et al., 2017; He et al., 2019). Finally, inducing stochasticity may help to escape from saddle points and local minima.*

Due to the statistical properties of Monte Carlo estimator,

$$\begin{aligned} \frac{1}{N_b} \sum_{i=1}^{N_b} L'(\boldsymbol{x}_{p(i)}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_{p(i)}, \boldsymbol{\theta}^k) &= \int L'(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N + O\Big(\frac{1}{\sqrt{N_b}}\Big). \\ &= \frac{1}{N} \sum_{i=1}^{N} L'(\boldsymbol{x}_i, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) + O\Big(\frac{1}{\sqrt{N_b}}\Big) \end{aligned}$$

Furthermore, asymptotically (when N_b and N grow), thanks to the central limit theorem,

$$\begin{aligned}\frac{1}{N_b}\sum_{i=1}^{N_b} L'(\boldsymbol{x}_{p(i)},\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_{p(i)},\boldsymbol{\theta}^k) &= \int L'(\boldsymbol{x},\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N + \frac{\boldsymbol{\Lambda}(\boldsymbol{\theta}^k)}{\sqrt{N_b}}\mathbf{g},\\ &= \frac{1}{N}\sum_{i=1}^{N} L'(\boldsymbol{x}_i,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i,\boldsymbol{\theta}^k) + \frac{\boldsymbol{\Lambda}(\boldsymbol{\theta}^k)}{\sqrt{N_b}}\mathbf{g}\end{aligned}$$

where $\mathbf{g}\sim\mathcal{N}(0,\boldsymbol{I}_n)$ and $\boldsymbol{\Lambda}(\boldsymbol{\theta}^k)$ is defined such that

$$\begin{aligned}[\boldsymbol{\Lambda}\boldsymbol{\Lambda}^T](\boldsymbol{\theta}^k) =& \int \big(L'(\boldsymbol{x},\boldsymbol{\theta})\big)^2 \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}^k}^T f(\boldsymbol{x},\boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N\\ &- \left[\int L'(\boldsymbol{x},\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N\right]\left[\int L'(\boldsymbol{x},\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}^k}^T f(\boldsymbol{x},\boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}^N\right],\\ =&\frac{1}{N}\sum_{i=1}^{N}\big(L'(\boldsymbol{x}_i,\boldsymbol{\theta})\big)^2\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i,\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i,\boldsymbol{\theta}^k)^T\\ &- \frac{1}{N^2}\sum_{p,q=1}^{N} L'(\boldsymbol{x}_p,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_p,\boldsymbol{\theta}^k) L'(\boldsymbol{x}_q,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}^k}^T f(\boldsymbol{x}_q,\boldsymbol{\theta}^k),\end{aligned}$$

where the matrix $[\boldsymbol{\Lambda}^k\boldsymbol{\Lambda}^{kT}]\in\mathbb{R}^{n\times n}$ is the covariance matrix of $\boldsymbol{\Lambda}^k\mathbf{g}$. Consequently, for one SGD iteration, we asymptotically have

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k-1} - \frac{\gamma}{N}\sum_{i=1}^{N} L'(\boldsymbol{x}_i,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i,\boldsymbol{\theta}^k) + \frac{\gamma}{\sqrt{N_b}}\boldsymbol{\Lambda}^k\mathbf{g}. \tag{5.16}$$

Remark. *In the previous derivations, we considered that the set $\{\boldsymbol{x}_{g(1)},...,\boldsymbol{x}_{g(N_b)}\}$ was sampled from $\{\boldsymbol{x}_1,...,\boldsymbol{x}_{N_b}\}$ with replacements. However, most of the time, $\{\boldsymbol{x}_{g(1)},...,\boldsymbol{x}_{g(N_b)}\}$ is sampled without replacement, so $K=N/N_b$ iterations are performed following equation (5.16). These K iterations are called an **epoch**. In fact, there are two loops in SGD. In the first, one iteration is one epoch during which all the points in the data set are used. The second loop consists of K updates of $\boldsymbol{\theta}$ and is nested in one epoch. In this section, we chose not to formalize the epochs since it would make the notations heavier without affecting the analysis.*

Equation (5.16) emphasizes that the optimization trajectory of $\boldsymbol{\theta}$ in SGD can be seen as a realization of a stochastic process. This observation naturally raises the question of how SGD integrates into our PDE framework. The answer relies on modeling choices for L, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^*-\boldsymbol{\theta}$ when comparing equation (5.16) with equation (5.15).

Reminder: The PDE framework for learning

In the PDE framework, learning consists of simulating the stochastic process

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \frac{\Delta t}{N}\sum_{i=1}^{N}\left[\boldsymbol{\alpha}L'(\boldsymbol{x}_i,\boldsymbol{\theta}^k) + L''(\boldsymbol{x}_i,\boldsymbol{\theta}^k)\boldsymbol{\alpha}\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i,\boldsymbol{\theta}^k)(\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right] + \frac{\sqrt{\Delta t}}{N}\left[\sum_{i=1}^{N}\boldsymbol{\Sigma}(\boldsymbol{x}_i,\boldsymbol{\theta}^k,\boldsymbol{\alpha},\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right]\mathbf{g}.$$

It asymptotically solves the PDE

$$\begin{cases} 0 = \partial_t \bar{f}(\boldsymbol{\theta}^t) \\ \quad + \sum_{i=1}^{n}\partial_{\theta_i}\bar{f}(\boldsymbol{\theta}^t)\int \alpha_i\left[L'(\boldsymbol{x},\boldsymbol{\theta}^t) + L''(\boldsymbol{x},\boldsymbol{\theta}^t)\sum_{j=1}^{n}\partial_{\theta_j}f(\boldsymbol{x},\boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t)\right]d\mathbb{P}_{\mathbf{x}} \\ \quad + \frac{1}{2}\sum_{i,j=1}^{n}\partial^2_{\theta_i,\theta_j}\bar{f}(\boldsymbol{\theta}^t)2\alpha_i\int L'(\boldsymbol{x},\boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t)d\mathbb{P}_{\mathbf{x}}, \\ f(\boldsymbol{x},\boldsymbol{\theta}^0) = f^0(\boldsymbol{\theta}), \quad \boldsymbol{\theta}\in\Theta, \quad t\in[0,T], \quad \mathbf{x}\sim d\mathbb{P}_{\mathbf{x}}, \end{cases}$$

which matches the PDE formulation of learning described in equation (5.6) if $\hat{f}\equiv 0$ and for a given $\boldsymbol{\alpha}$. The expression of the drift term and the diffusion coefficients that are to be identified are:

- The drift term: $\boldsymbol{\mu}(\boldsymbol{\theta}) = \int \boldsymbol{\alpha}\Big(L'(\boldsymbol{x}_i,\boldsymbol{\theta}) + L''(\boldsymbol{x}_i,\boldsymbol{\theta})\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x},\boldsymbol{\theta})(\boldsymbol{\theta}^* - \boldsymbol{\theta})\Big)d\mathbb{P}_{\mathbf{x}}$,
- The diffusion coefficients: $\left[\boldsymbol{\Sigma}\boldsymbol{\Sigma}^T\right]_{i,j}(\boldsymbol{\theta}) = 2\int \alpha_i L'(\boldsymbol{x},\boldsymbol{\theta})(\theta_j^* - \theta_j)d\mathbb{P}_{\mathbf{x}}$.

For readability, we provide a reminder in 5.4.1 to emphasize the importance of choosing L, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}$. Let us choose $\boldsymbol{\alpha} = -\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x},\boldsymbol{\theta}^k)$. We retrieve the first term of $\boldsymbol{\mu}$, the drift term. Then, the second term in the drift becomes

$$\alpha_i L''(\boldsymbol{x},\boldsymbol{\theta}^t)\partial_{\theta_j}f(\boldsymbol{x},\boldsymbol{\theta}^t)(\theta_j^* - \theta_j^t) = \frac{1}{N}\sum_{i=1}^{N}L''(\boldsymbol{x}_i,\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}}f(\boldsymbol{x}_i,\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x},\boldsymbol{\theta})(\boldsymbol{\theta}^* - \boldsymbol{\theta}^k). \quad (5.17)$$

In order to identify equation (5.16) with equation (5.15), we have to make sure that this term is negligible. This happens if either $\boldsymbol{\theta}^* - \boldsymbol{\theta} = 0$ - but it would cancel the diffusion coefficients thereby forbidding stochasticity - or if $L'' = 0$. We then have to choose the second solution and set $L'' = 0$, which implies that L' is a smooth L^1 norm.

Let us now focus on the diffusion coefficients. Considering the previous choices, we have to ensure that

$$2\int L'(\boldsymbol{x},\boldsymbol{\theta}^k)\partial_{\theta_i}f(\boldsymbol{x},\boldsymbol{\theta}^k)(\theta_j^*-\theta_j^k)d\mathbb{P}_{\mathbf{x}} = \gamma^2\int \left(L'(\boldsymbol{x},\boldsymbol{\theta})\right)^2\partial_{\theta_i}f(\boldsymbol{x},\boldsymbol{\theta}^k)\partial_{\theta_j}f(\boldsymbol{x},\boldsymbol{\theta}^k)^T d\mathbb{P}_{\mathbf{x}} - \gamma^2\int\int L'(\boldsymbol{x},\boldsymbol{\theta})\partial_{\theta_i}f(\boldsymbol{x},\boldsymbol{\theta}^k)L'(\boldsymbol{x}',\boldsymbol{\theta})\partial_{\theta_j}f(\boldsymbol{x}',\boldsymbol{\theta}^k)d\mathbb{P}_{\mathbf{x}}d\mathbb{P}_{\mathbf{x}'},$$

where $\mathbf{x}'$ follows the same distribution of $\mathbf{x}$. This condition holds if we choose:

$$\boldsymbol{\theta}^*-\boldsymbol{\theta}^k = \frac{\gamma^2}{2}L'(\boldsymbol{x},\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x},\boldsymbol{\theta}^k) - \frac{\gamma^2}{2}\int L'(\boldsymbol{x},\boldsymbol{\theta}^k)\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x},\boldsymbol{\theta}^k)d\mathbb{P}_{\mathbf{x}}.$$

By making hypotheses on $\boldsymbol{\alpha}$, L, and $\boldsymbol{\theta}^*-\boldsymbol{\theta}$, we managed to recast SGD as a special case of our PDE framework.

5.4.2 A PDE based SGD

Considering SGD under the previously described PDE framework implies modeling choices that may not be reasonable. In this part, we construct a modified version of SGD, which we call PDESGD, which enjoys benefits from the PDE framework.

5.4.2.1 Alleviating constraints on the stochastic process

In the previous section, we demonstrated that under some hypothesis made on $\boldsymbol{\alpha}$, L and $\boldsymbol{\theta}^*-\boldsymbol{\theta}$, the process $\boldsymbol{\theta}^k$ simulated with SGD solves the PDE equation (5.10). However, these hypothesis hold under constraints on L and $\hat{f}$. These constraints are :

- $\hat{f}\sim 0$,
- $L''\sim 0$.

For machine learning problems where $L''\sim 0$, e.g. when the loss function is a smooth L_1 error, SGD is known to yield satisfying results (this loss function is implemented in all deep learning libraries). Hence, in this work, we assume that the constraint $\hat{f}\sim 0$ is reasonable (this point is thoroughly discussed in Poëtte et al. (2021)). However, the constraint $L''\sim 0$ is not respected in many machine learning problems. It is the case, for instance, when L is chosen as the mean squared error or the binary cross-entropy. In this section, we make different choices for $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^*-\boldsymbol{\theta}$ that alleviate this constraint, resulting in a new update formula that is slightly modified compared to that of the classical SGD.

Let us choose

$$
\begin{aligned}
\boldsymbol{\alpha} =& -\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k), \\
\boldsymbol{\theta}^* - \boldsymbol{\theta} =& \frac{\lambda}{2} L'(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) - \frac{\lambda}{2} \int L'(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}}, \\
\boldsymbol{\mu}(\boldsymbol{\theta}^t) =& -\int \Big[L'(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) + \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \\
& - \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \int L'(\boldsymbol{x}', \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}', \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}'} \Big] d\mathbb{P}_{\mathbf{x}}, \\
\boldsymbol{\Sigma}(\boldsymbol{\theta}^t) =& \frac{1}{\sqrt{N_b}} \boldsymbol{\Lambda}(\boldsymbol{\theta}^t),
\end{aligned}
$$

where $\boldsymbol{\Lambda}(\boldsymbol{\theta}^t)$ is given by

$$
\begin{aligned}
[\boldsymbol{\Lambda}\boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t) =& \frac{\lambda}{N} \sum_{i=1}^{N} \big(L'(\boldsymbol{x}_i, \boldsymbol{\theta}) \big)^2 \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k)^T \\
& - \frac{\lambda}{N^2} \sum_{p,q=1}^{N} L'(\boldsymbol{x}_p, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_p, \boldsymbol{\theta}^k) L'(\boldsymbol{x}_q, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}^k}^T f(\boldsymbol{x}_q, \boldsymbol{\theta}^k),
\end{aligned}
\tag{5.18}
$$

and $\lambda \in [0, 1]$ is a parameter whose value is fixed to 1 for now. These choices are similar to those of SGD, except that we did not suppose that $L'' \sim 0$. Moreover, in our case, the noise is explicitly included in the update, as opposed to SGD, where the noise comes from sampling batches. The noise can be controlled by simulating an arbitrary batch size of N_b, even if the gradient is estimated with $N > N_b$ points.

The iteration formula then becomes

$$
\begin{aligned}
\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\Delta t}{N} \sum_{i=1}^{N} \Big[& L'(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \\
& + \frac{\lambda}{2} L''(\boldsymbol{x}_i, \boldsymbol{\theta}^k) L'(\boldsymbol{x}_i, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \\
& - \frac{\lambda}{2N} L''(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \sum_{j=1}^{N} L'(\boldsymbol{x}_j, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_j, \boldsymbol{\theta}^k) \Big] \\
& + \sqrt{\frac{\Delta t}{N_b}} \boldsymbol{\Lambda}(\boldsymbol{\theta}^t) \mathbf{g}.
\end{aligned}
\tag{5.19}
$$

Note that we do not have to find an explicit expression of $\boldsymbol{\Lambda}(\boldsymbol{\theta}^t)$ in order to simulate this process. It is enough sampling from a centered multivariate Gaussian random variable of dimension n, and of covariance matrix $[\boldsymbol{\Lambda}\boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t)$, which can be estimated. Averaging realizations of this stochastic process statistically solves the PDE system :

$$
\begin{cases}
0 = \ \partial_t \bar{f}(\boldsymbol{\theta}^t) \\
\quad - \sum_{i=1}^{n} \partial_{\theta_i} \bar{f}(\boldsymbol{\theta}^t) \int \Big[L'(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \\
\qquad\qquad + \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \\
\qquad\qquad - \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \int L'(\boldsymbol{x}', \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}', \boldsymbol{\theta}^k) d\mathbb{P}_{\mathbf{x}'} \Big] d\mathbb{P}_{\mathbf{x}} \\
\quad + \frac{1}{N_b} \sum_{i,j=1}^{n} \partial^2_{\theta_i, \theta_j} \bar{f}(\boldsymbol{\theta}^t) [\boldsymbol{\Lambda} \boldsymbol{\Lambda}^T]_{i,j}(\boldsymbol{\theta}^t), \\
f(\boldsymbol{x}, \boldsymbol{\theta}^0) = f^0(\boldsymbol{\theta}), \ \ \boldsymbol{\theta} \in \Theta, \ \ t \in [0, T], \ \ \mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}.
\end{cases}
\tag{5.20}
$$

With these choices, averaging realizations of the process described in equation (5.19) solves equation (5.20), which is equivalent to solving equation (5.1) under the only constraint that $\hat{f} \equiv 0$. The constraint is less stringent than for SGD, and we can use any loss function. We call the optimization algorithm defined by the parameter update of equation (5.19) PDESGD.

5.4.2.2 Stability of the process

Classical results from ODE and PDE theory state that at iteration k, taking a time step $\Delta^k t$ of the form

$$
\Delta^k t = \times \min \left(\frac{\epsilon}{\max_{i \in \{1, \dots, n\}} \mu_i(t^k, \boldsymbol{\theta}^k)}, \frac{\epsilon^2}{\max_{i \in \{1, \dots, n\}} [\Sigma(t^k, \boldsymbol{\theta}^k) \mathbf{g}]_i^2} \right),
\tag{5.21}
$$

ensures having stability and accuracy $\mathcal{O}(\epsilon)$ for process $(\boldsymbol{\theta}^k)_{k \in \mathbb{N}}$ in the approximation of $(\boldsymbol{\theta}^t)_{t \in \mathbb{R}^+}$. In other words, Equation (5.21) gives a value for the time step, or the learning rate, that theoretically ensures stability and approximation accuracy of $\mathcal{O}(\epsilon)$.

5.4.2.3 Complexity of PDESGD

The iteration formula of PDESGD introduces additional operations in comparison to SGD. The additional computations involved in $\boldsymbol{\mu}(\boldsymbol{\theta}^k)$ can be implemented such that their complexity is $\mathcal{O}(n)$, when the vector products are computed from right to left, which is the same complexity as one iteration of SGD. However, computing $[\boldsymbol{\Lambda} \boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t)$ to sample from $\boldsymbol{\Lambda}(\boldsymbol{\theta}^t)\mathbf{g}$ has complexity $\mathcal{O}(n^2)$. Hence, in the following, we approximate $[\boldsymbol{\Lambda} \boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t)$ with $diag\big([\boldsymbol{\Lambda} \boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t)\big)$ so that computing $[\boldsymbol{\Lambda} \boldsymbol{\Lambda}^T](\boldsymbol{\theta}^t)$ has complexity $\mathcal{O}(n)$ and the complexity of PDESGD stays the same as SGD.

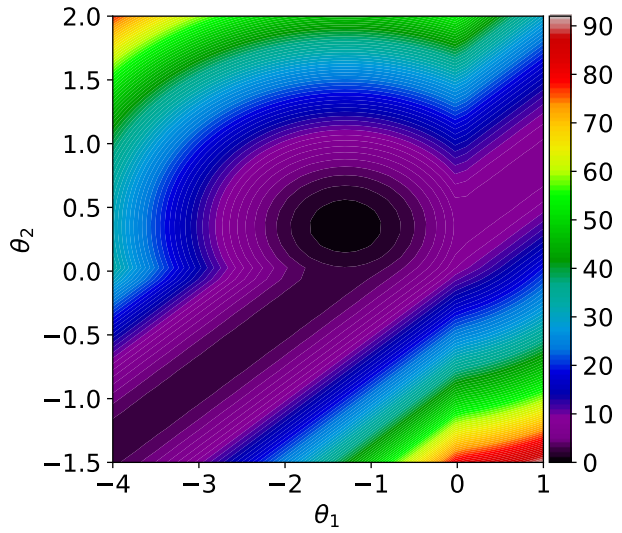


Figure 5.1: The loss function $\hat{J}$ plotted with respect to θ_1 and θ_2

5.4.3 Toy experiments

We test the algorithm obtained using our PDE framework on a simplistic optimization problem in two dimensions. The problem consists in optimizing two weights of a neural network in the approximation of the constant function $x \to 1$ using $N = 100$ points $\{x_1, ..., x_N\}$ equidistant between -5 and 5. The neural network has the following characteristics :

- `n_layers` $= 1$,
- `n_units` $= 2$,
- `activation` $= \sigma =$ Relu,
- `loss_function` $= L_2$ error,
- has no bias parameters
- has a residual connection between the input and the output (like in Resnet, see He et al. (2015)).

The parameters to optimize are $\boldsymbol{\theta} = \{\theta_1, \theta_2\}$, the weights of the first layer, and $f_{\boldsymbol{\theta}}$ can be written :

$$f_{\boldsymbol{\theta}}(x) = \sigma(\theta_1 x) - 2\sigma(\theta_2 x) + x,$$

so the objective function that we aim at minimizing is:

$$\hat{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\left(\sigma(\theta_1 x_i) - 2\sigma(\theta_2 x_i) + x_i - 1\right)^2 \approx J(\boldsymbol{\theta}) = \frac{1}{10}\int_{-5}^{5}\left(\sigma(\theta_1 x) - 2\sigma(\theta_2 x) + x - 1\right)^2 d\boldsymbol{x}.$$

The loss landscape $(\theta_1, \theta_2) \to \hat{J}(\theta_1, \theta_2)$ is represented in Figure 5.1. We select this test case as a toy example for two reasons. First, its dimensionality makes it possible to visualize the loss landscape precisely. Some works aim at giving a 2D visualization of the surface when $Card(\boldsymbol{\theta}) > 2$ using projections (Li et al., 2018a), but the obtained surface is always an approximation. In our case, we would like to track all the local minima and to be able to assess whether the tested algorithms reach the vicinities of the global minimum or not. The second reason why we choose this test case is that there is one remarkable global minimum and an infinity of local minima. In the following, we test the effect of the iterative algorithm with the update formula given by equation (5.19). We also explore the benefits of the stability granted by a learning rate given by equation (5.21).

5.4.3.1 PDESGD in a convex setting

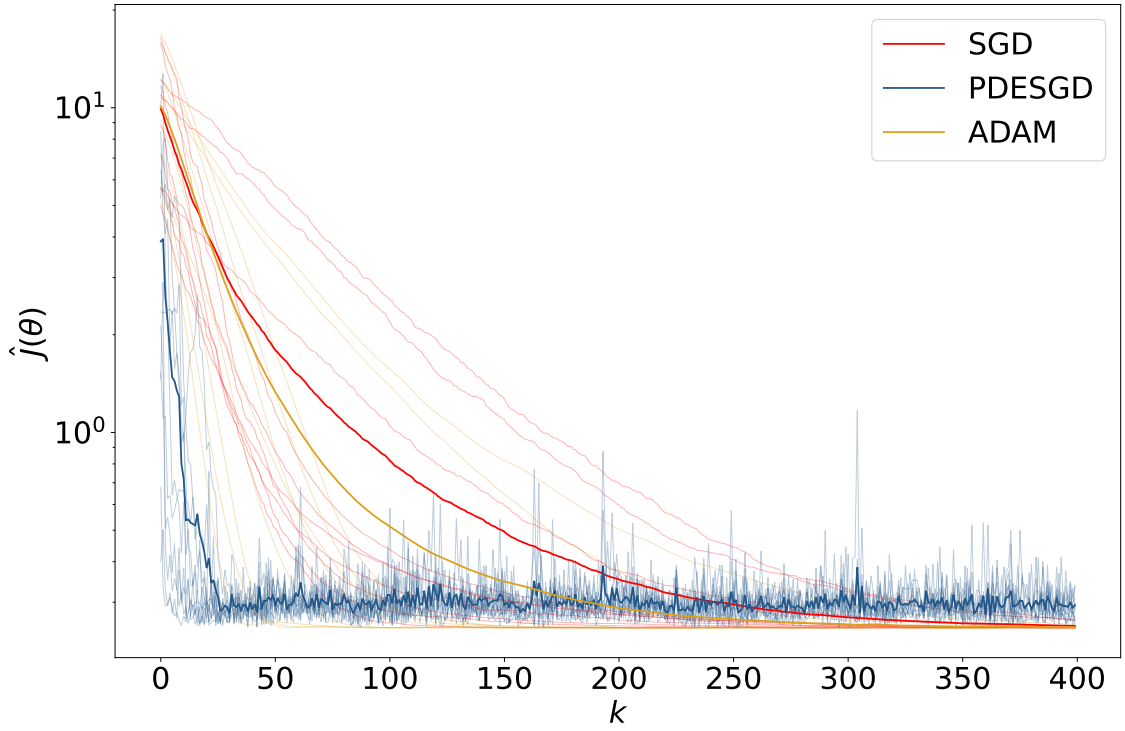


Figure 5.2: The loss function J plotted with respect to the number of iterations of the optimization process.

In this experiment, we test the convergence speed of PDESGD when the process starts in a convex area around a minimum, whether it is local or global. To that end, we randomly initialize 10 processes in the square $[-3, -0.5] \times [0.1, 1.5]$ and track the value of $J(\boldsymbol{\theta})$ along the iterations. We also conduct the same experiment with SGD and another popular optimization

algorithm, Adam (Kingma and Ba, 2015). For PDESGD and SGD, we use the same learning rate of 10^{-3} to be able to compare the trajectories (otherwise, the speed of the algorithm with the highest learning rate would be boosted). Note that using this learning rate for Adam makes it so much slower than SGD and PDE-consistent SGD that we took the liberty of increasing it to 10^{-2}, even if 10^{-3} is its default value in many implementations. For SGD and ADAM, the batch size is 10, and for PDESGD, the actual batch size is $N = 100$, but we choose $N_b = 10$ to simulate the stochasticity coming from a batch size value of 10.

In Figure 5.2, we plot the 10 evolutions of $J(\boldsymbol{\theta})$ with respect to k and their mean (in bold) for each algorithm. First, PDESGD converges much faster than Adam and SGD, which is a good point. Then, two more observations may be related to one another. The first is that the curve of PDESGD is noisier than the others. The second is that PDESGD does not seem to converge towards the minimum $J(\boldsymbol{\theta})$ and stays stuck around a value slightly above. These observations can be explained by the stochasticity coming from $\mathbf{g}$, which prevents the process from sticking to the minimum.

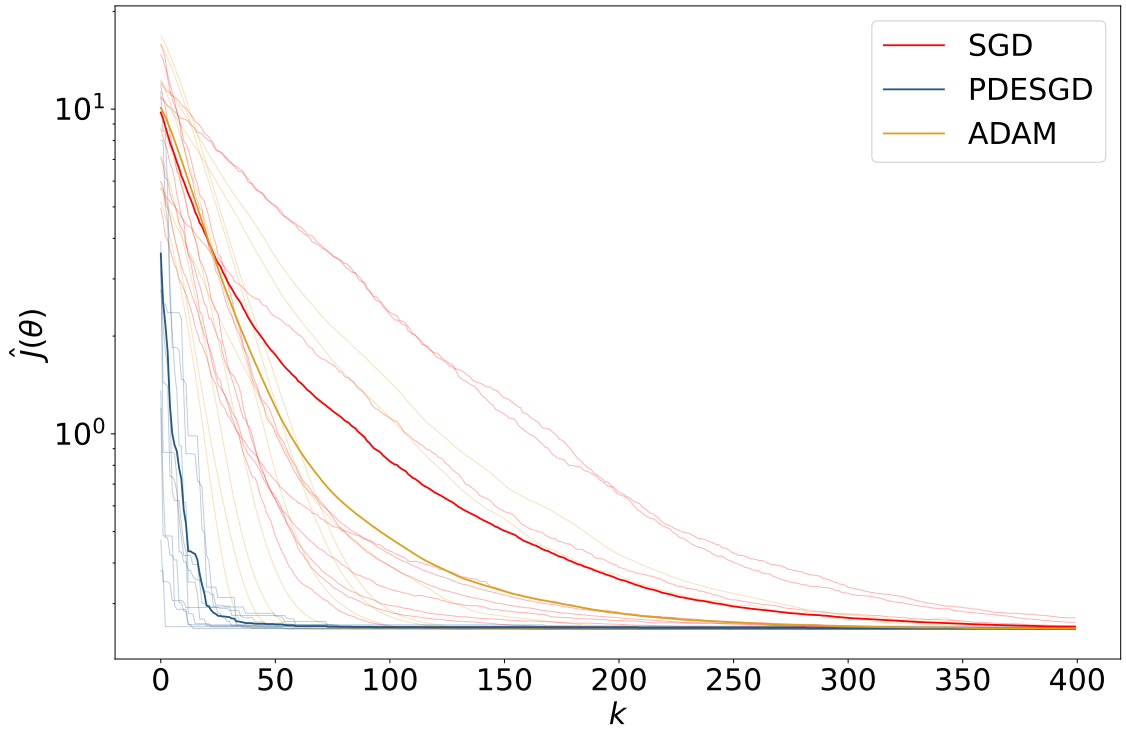


Figure 5.3: $\min_{i<k} J(\boldsymbol{\theta}^i)$ plotted with respect to k, the number of iterations of the optimization process.

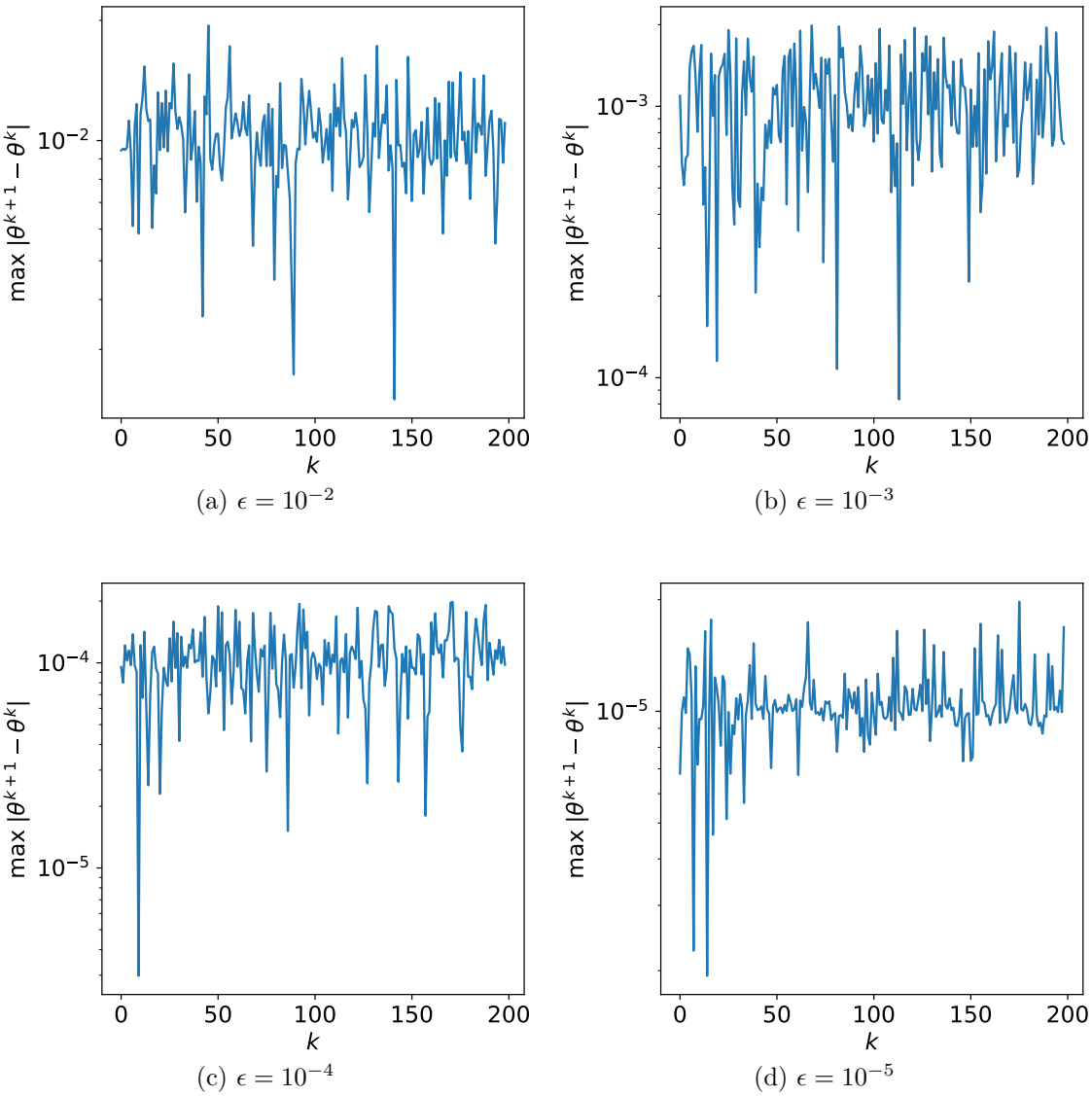
It should be recalled that PDESGD is constructed as an exploratory process as well as an optimization process. Moreover, the optimization result is the value of $\boldsymbol{\theta}^k$ that led to the minimum of $J(\boldsymbol{\theta})$ throughout all the optimization trajectories rather than the last value of $\boldsymbol{\theta}^k$. To illustrate this point, in Figure 5.3, we plot the evolution of $\min_{i<k} J(\boldsymbol{\theta}^i)$ for the 10 different initializations with respect to k for each algorithm.

This experiment shows that making SGD consistent with the PDE framework, alleviating

the constraint $L'' = 0$, grants a significant convergence speed-up to the optimization process. It also emphasizes the nuance between exploration and optimization and how they may be related.

5.4.3.2 Effect of the constrained learning rate

In this section, we illustrate the effect of the learning rate defined in equation (5.21). We run PDESGD for 200 iterations and monitor $\max(|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|)$, for $\epsilon \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$, where $\max(|\boldsymbol{x}|)$ denotes the highest component of the absolute value of $\boldsymbol{x}$. We plot $\max\ |\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|$ with respect to k in Figure 5.4 for each value of ϵ.



(a) $\epsilon = 10^{-2}$ (b) $\epsilon = 10^{-3}$

(c) $\epsilon = 10^{-4}$ (d) $\epsilon = 10^{-5}$

Figure 5.4: Plots of $\max(|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|)$ with respect to k for $\epsilon \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$

We recover the properties of accuracy ensured by equation (5.21). The advantages of being able to toggle the magnitude of $|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|$ for reaching the desired accuracy are twofold. First, we control the optimizer error, mentioned in 2.4.1. It could be a good asset for research studies looking to assess the importance of the different sources of error in deep learning. Second, it tackles a common problem in deep learning, identified by Bengio et al. (1994) called the exploding gradient, whereby the gradients can suddenly take high values, yielding an unstable optimization. Our solution is different than the traditional gradient clipping method Pascanu et al. (2013) in the sense that we impose a constraint on $|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|$ rather than $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

5.4.3.3 Constrained learning rate for stable exploration

Since deep learning is a non-convex optimization problem with high dimensions and plenty of local minima, it is desirable to explore the parameter space as much as possible. To that end, one solution is to increase the learning rate of the chosen optimization algorithm. However, by doing so, there is a risk of observing unstable and diverging optimization trajectories, and finding the good trade-off value for the learning rate may require many trials and errors.

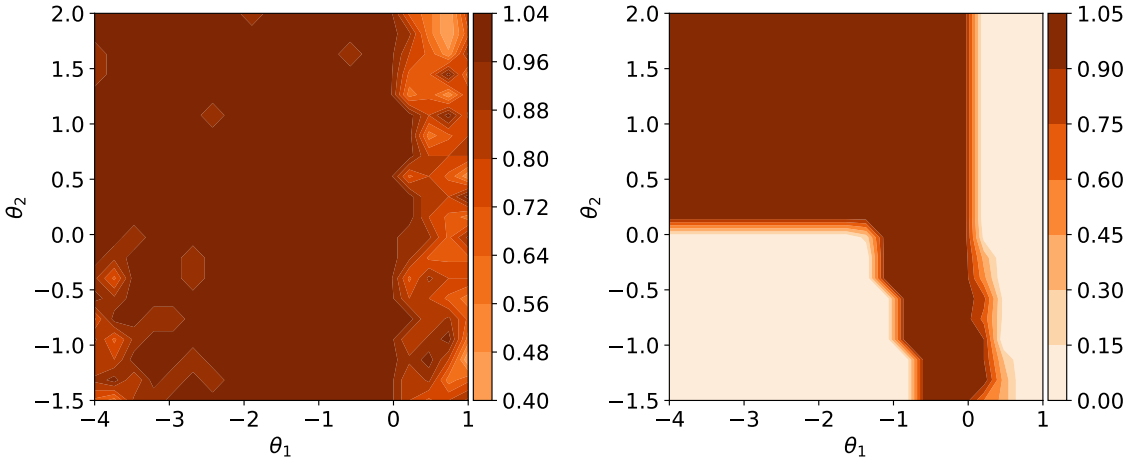
The learning rate constraint can be used for purposes other than those described in the previous section. Here, we highlight how we can use equation (5.21) to explore the parameter space. The exploration aims at looking for good potential convex regions in the parameter space. Hence, we define an objective convex region around the global minimum $\boldsymbol{\theta}^*$, which we exactly identify using a Newton algorithm. This region is a 0.6×0.4 rectangle centered on the global minimum of $\hat{J}(\boldsymbol{\theta})$.

We initialize PDESGD and SGD processes on a 20×20 grid with $\theta_1 \in [-4, 1]$ and $\theta_2 \in [-1.5, 2]$, and evaluate the probability to reach the objective region from each point of the grid. For PDESGD, we choose $\epsilon = 1$, i.e. we have

$$\Delta^k t = \min \left(\frac{1}{\max\limits_{i \in \{1,\dots,n\}} \mu_i(t^k, \boldsymbol{\theta}^k)}, \frac{1}{\max\limits_{i \in \{1,\dots,n\}} [\Sigma(t^k, \boldsymbol{\theta}^k)\mathbf{g}]_i^2} \right). \tag{5.22}$$

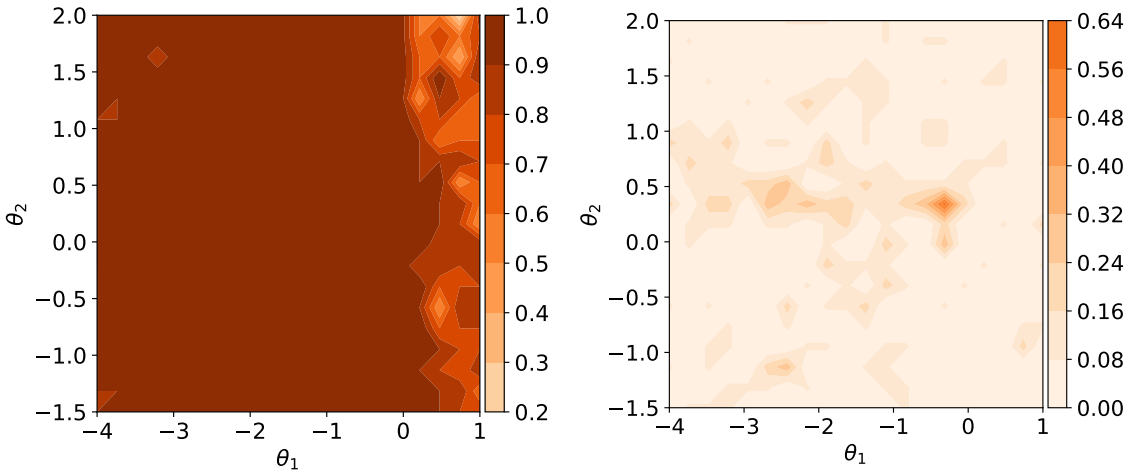
For SGD, we reproduce this experiment for $\gamma \in \{0.01, 0.05, 0.1\}$. To have a visual overview of the exploration, we plot a heat map whose pixels value is the probability of the considered algorithm reaching the region of interest when initialized on this point. This probability is evaluated in each point with 10 different random seeds. The evaluation is coarse, but this is not so much of a problem here since the primary goal is to illustrate the exploration behavior of the algorithms. The heat maps can be found in Figure 5.5. We also compute the probability to reach the region of interest integrated over the grid. The results are gathered in Table 5.1.

These results show that PDESGD, when coupled with learning rate limitation, explores the domain and almost always finds the convex region. For SGD, a learning rate tuning is required:



(a) PDESGD. $P = 0.94$

(b) SGD, $\gamma = 0.01$, $P = 0.55$



(c) SGD, $\gamma = 0.05$, $P = 0.95$

(d) SGD, $\gamma = 0.1$, $P = 0.05$

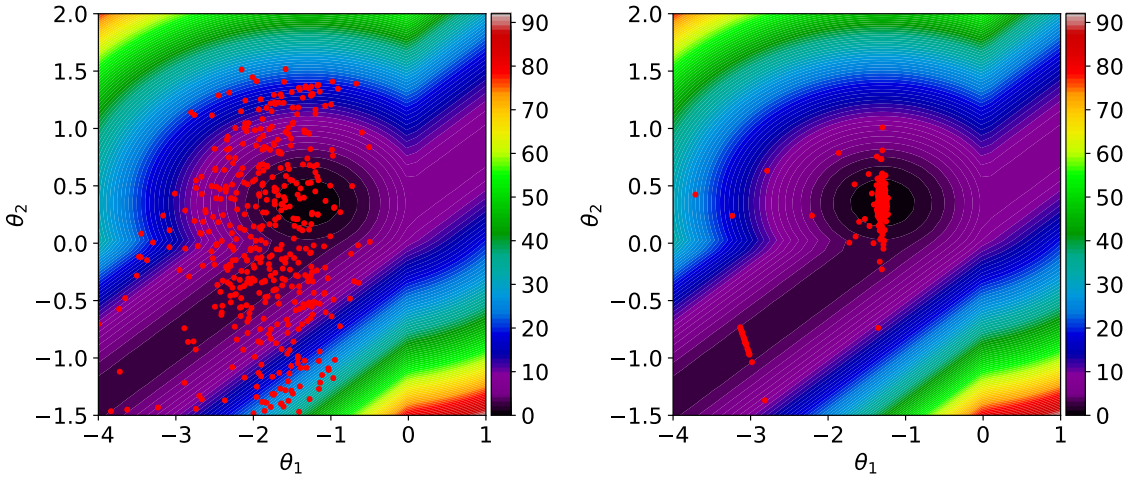
Figure 5.5: Heat maps of the probability to reach the region of the global minimum from coordinates of the map for PDESGD and SGD with different learning rates γ. The total probability P is also displayed.

	PDESGD	SGD		
γ	-	0.01	0.05	0.1
P	0.94	0.55	0.95	0.05

Table 5.1: Probability to reach the region of interest

$\gamma = 0.05$ yields result comparable with those of PDESGD, but if $\gamma = 0.01$, SGD is far less likely to find the convex region when initialized near suboptimal local minima and when $\gamma = 0.1$, SGD is unstable and almost always diverges. It illustrates the practical advantage of having a stable process without carefully tuning optimization hyperparameters. We also plot

the points visited by the process on Figure 5.6, for PDESGD and SGD with $\gamma = 0.05$, when $\boldsymbol{\theta}^0 = (-3, -1)$ (outside the convex region). It shows the differences in the behavior of SGD and PDESGD. The process of PDESGD is more spread and visits the vicinities of many local minima and the global minimum, while SGD is more focused on the global minimum. It is natural since SGD is not constructed as an exploration algorithm.



(a) PDESGD (b) SGD

Figure 5.6: Trajectories of the process of PDESGD and SGD for an initialization of $\boldsymbol{\theta}^0 = (-3, -1)$, outside the convex region.

Still, as such, this process is not sufficient to complete the optimization process. As showed in equation (5.21), using equation (5.22) to compute the step size only ensures an accuracy for $\boldsymbol{\theta}^k$ which is $\mathcal{O}(1)$. Hence, this algorithm is only suited for the exploration of the parameter space and not the optimization of the objective function.

5.4.3.4 Bias of PDESGD

Despite all those promising results, PDESGD as such has limitations. Indeed, it turns out that in our test case, in the convex region, the point towards which the algorithms converges is not the global minimum but a point that is slightly shifted. This phenomenon is illustrated in Figure 5.7, where we initialized a PDESGD process without noise, without learning rate constraint, and with $\lambda = 10^{-4}$, near the global minimum.

This bias in the obtained minimum may be explained by the second and the third terms in $\mu(\boldsymbol{\theta}^t)$. Let us come back to GD. If $\gamma \neq 0$, the convergence of the process is reached if and only if $||\boldsymbol{\mu}(\boldsymbol{\theta}^t)|| = \gamma||\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t)|| < \epsilon'$, with $\epsilon' > 0$ a convergence criterion. In other words:

$$||\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|| < \epsilon' \Leftrightarrow ||\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k)|| < \frac{\epsilon'}{\gamma},$$

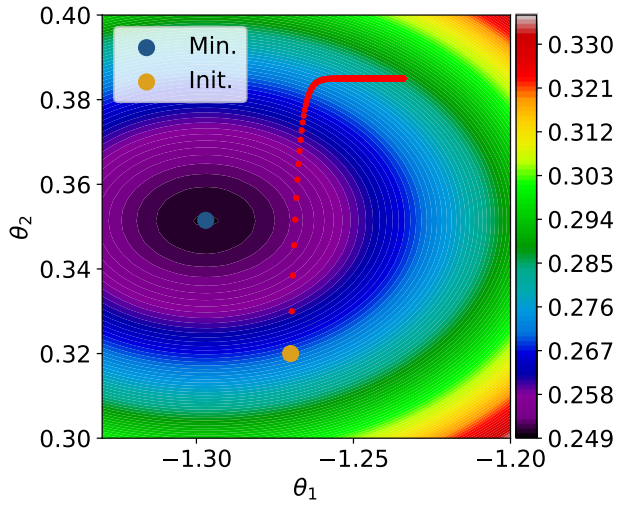


Figure 5.7: Trajectory of the optimization near the global minimum.

so canceling the gradient $||\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k)||$ with accuracy $\frac{\epsilon'}{\gamma}$ is equivalent to reaching $||\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|| < \epsilon'$. In our case, this equivalency does not hold. Indeed, recall the expression of $\mu(\boldsymbol{\theta}^t)$ for PDESGD:

$$\begin{aligned}\boldsymbol{\mu}(\boldsymbol{\theta}^t) = -\int \Big[& L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) + \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^t) L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \\ & - \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) \int L'(\boldsymbol{x}', \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}', \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}'} \Big] d\mathbb{P}_{\mathbf{x}}.\end{aligned}$$

Then, by noting that $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k) = \int L'(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k)$

$$\begin{aligned}||\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k|| < & \epsilon' \\ \Leftrightarrow & ||\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k) + \int \Big[\frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) L'(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \\ & - \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^k) \Big] d\mathbb{P}_{\mathbf{x}} || < \epsilon'.\end{aligned}$$

Therefore, it is far from obvious that the convergence of the process is equivalent to the cancellation of the gradient in this case because the terms of the left-hand side could cancel each other without the gradient being equal to zero.

5.4.4 Unbiasing PDESGD and experiments on real-world datasets

This section introduces a different choice for $\boldsymbol{\theta}^* - \boldsymbol{\theta}$ that solves the bias problem previously described, when L is the L_2 error. We demonstrate how it circumvents this problem and showcase examples of this "unbiased" PDESGD on real-world regression tasks.

5.4.4.1 Unbiasing PDESGD

Let us recall the previous choice for $\boldsymbol{\theta}^* - \boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* - \boldsymbol{\theta} = \frac{\lambda}{2} L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) - \frac{\lambda}{2} \int L'(\boldsymbol{x}_i, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}}.$$

Now, let us consider another choice:

$$\boldsymbol{\theta}^* - \boldsymbol{\theta} = \frac{\lambda}{2} \int L'(\boldsymbol{x}_i, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}} = \frac{\lambda}{2} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t), \tag{5.23}$$

where we only kept the right part of $\boldsymbol{\theta}^* - \boldsymbol{\theta}$. In that case, the expression for $\mu(\boldsymbol{\theta}^t)$ becomes:

$$\boldsymbol{\mu}(\boldsymbol{\theta}^t) = -\int \Big[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) + \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) \Big] d\mathbb{P}_{\mathbf{x}}. \tag{5.24}$$

The intuition behind this choice stems from the following lemma, that ensures that for this choice of $\boldsymbol{\theta}^* - \boldsymbol{\theta}$, cancelling $\boldsymbol{\mu}(\boldsymbol{\theta}^t)$ is equivalent to cancelling the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t)$, unlike for the previous version of PDESGD:

Lemma 5. *Let $\boldsymbol{\theta}^* - \boldsymbol{\theta}$ be chosen as in equation (5.23), $\boldsymbol{\mu}(\boldsymbol{\theta}^t)$ defined as in equation (5.24), and L chosen as the L_2 error. Then,*

$$\boldsymbol{\mu}(\boldsymbol{\theta}^t) = 0 \Leftrightarrow \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) = 0.$$

Proof. Let us come back to equation (5.24):

$$\begin{aligned} -\boldsymbol{\mu}(\boldsymbol{\theta}^t) &= \int \Big[L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) + \frac{\lambda}{2} L''(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) \Big] d\mathbb{P}_{\mathbf{x}}. \\ &= \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) + \int \Big[\lambda \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) \Big] d\mathbb{P}_{\mathbf{x}}. \end{aligned}$$

since $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) = \int L'(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}}$ and $L''(\boldsymbol{x}, \boldsymbol{\theta}^t) = 2$. Then,

$$\begin{aligned} -\boldsymbol{\mu}(\boldsymbol{\theta}^t) &= \left(\boldsymbol{I}_n + \lambda \left[\int \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f^T(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}} \right] \right) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t), \\ -\boldsymbol{\mu}(\boldsymbol{\theta}^t) &= F(\boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t), \end{aligned}$$

where $F(\boldsymbol{\theta}^t) = \boldsymbol{I}_n + \lambda \Big[\int \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f^T(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}} \Big]$. By definition, $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f^T(\boldsymbol{x}, \boldsymbol{\theta}^t)$

is positive semidefinite, so $\int \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} f^T(\boldsymbol{x}, \boldsymbol{\theta}^t) d\mathbb{P}_{\mathbf{x}}$ is positive semidefinite as well. Since $\boldsymbol{I}_n$ is positive definite, so is $F(\boldsymbol{\theta}^t)$, which implies that it only has non zero positive eigenvalues:

$$\boldsymbol{\mu}(\boldsymbol{\theta}^t) = 0 \Leftrightarrow F(\boldsymbol{\theta}^t) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) = 0 \Leftrightarrow \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^t) = 0.$$

□

Therefore, in the following, we use PDESGD on regression problems based on L_2 error defined by equation (5.23) and equation (5.24) with the following iteration formula:

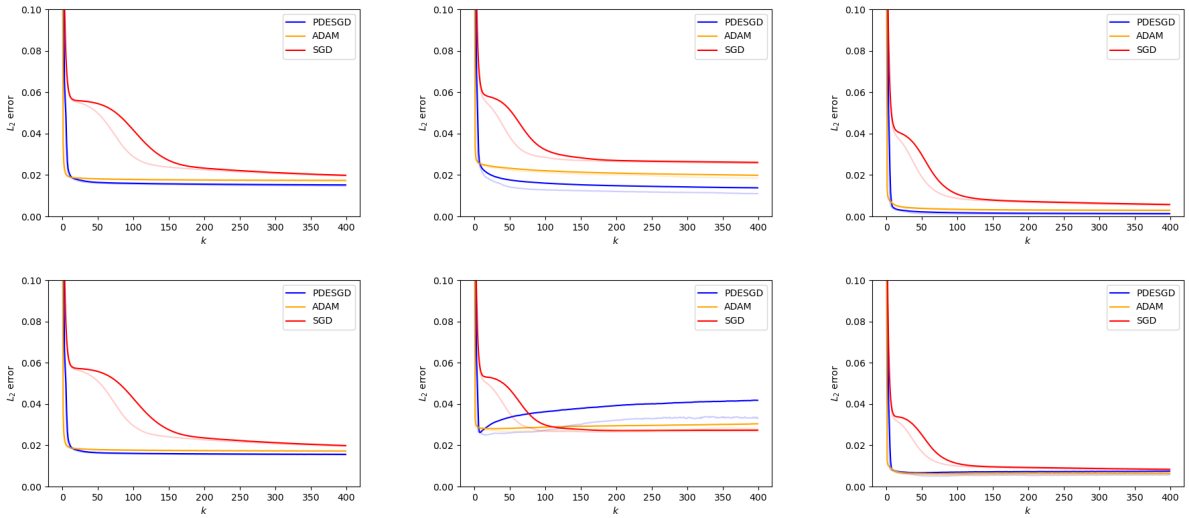
$$\boxed{\begin{aligned} \boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \frac{\Delta t}{N} \sum_{i=1}^{N} \Big[& L'(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \\ & + \frac{\lambda}{N} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i, \boldsymbol{\theta}^k) \sum_{j=1}^{N} L'(\boldsymbol{x}_j, \boldsymbol{\theta}^k) \nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}_j, \boldsymbol{\theta}^k) \Big] \\ & + \sqrt{\frac{\Delta t}{N_b}} \boldsymbol{\Lambda}(\boldsymbol{\theta}^k) \mathbf{g}. \end{aligned}} \tag{5.25}$$

5.4.4.2 Experiments on real-world datasets

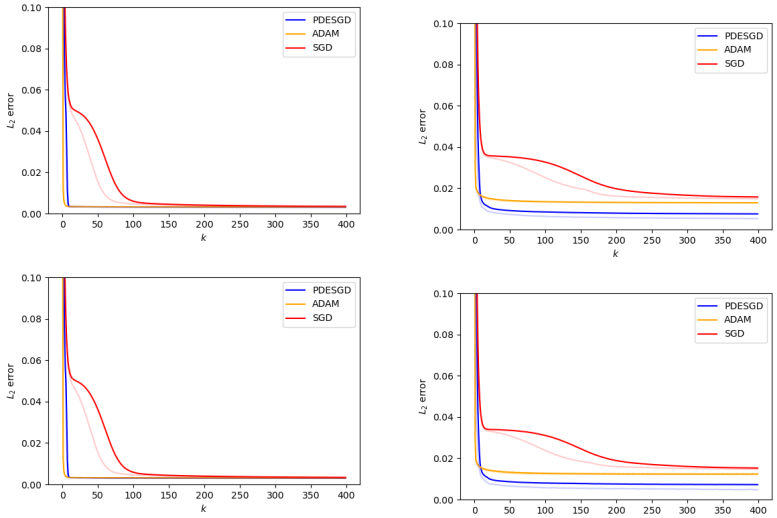
In this section, we test PDESGD on five real-world regression datasets extracted from UCI dataset repository (Dua and Graff, 2017). We consider California Housing (CH), Diabetes (D), Boston Housing (BH), Combined Cycle Power Plant (CCPP), and Airfoil Self Noise (ASN) datasets. An MLP with one hidden layer of 10 units, with ReLU activation function is trained with the L_2 error as loss function using SGD, ADAM and PDESGD with a learning rate of 10^{-3} on $k_f = 10^6$ steps. For SGD and ADAM, the batch size is 50, 10, 10, 50 and 20 for CH, D, BH, CCPP and ASN respectively. For PDESGD, the actual batch size is $\min(1000, N)$, and N_b is set to the corresponding value of the batch size of SGD and ADAM. The dataset is randomly split between a training and validation set. For each dataset, $N_t = 30$ instances of training are executed with different initializations.

Figure 5.8 gathers the mean of the learning curves (in bold) as well as the curve that obtained the best validation error (in transparent) for each dataset and for the first 400 iterations. Table 5.2 displays the errors for PDESGD, ADAM and SGD. Let $\{L(\boldsymbol{\theta}_i^k)\}_{k \in \{1,..,k_f\}}$ be the learning curve of the i-th optimization. The format of the reported error is **best, mean**, where **best** $= \min_{k,i \in \{1,..,k_f\},\{1,..,N_t\}} L(\boldsymbol{\theta}_i^k)$, **mean** $= \frac{1}{N_t} \sum_{i=1}^{N_t} \min_{k \in \{1,..,k_f\}} L(\boldsymbol{\theta}_i^k)$. The standard error for the estimation of **mean** is not displayed because in our case it is always of order $< 10^{-6}$. Hence, the results of Table 5.2 are statistically significant.

Two observations can be made regarding these results:



(a) California Housing (CH) (b) Diabetes (D) (c) Boston Housing (BH)



(d) Combined Cycle Power Plant (CCPP) (e) Airfoil Self Noise (ASN)

Figure 5.8: Train (top) and validation (bottom) errors for each dataset, with respect to k, the number of $\boldsymbol{\theta}^k$ update. The bold curve is the mean of the 30 learning curves, and the transparent curve is the best of the 30 learning curves.

- **Convergence speed:** Figure 5.8 shows that PDESGD seems to exhibit a similar convergence speed as ADAM: both algorithms are significantly quicker than SGD. This is a good practical advantage for PDESGD.
- **Validation error:** Table 5.2 shows that, most of the time, the validation error of SGD ends up being lower than that of ADAM. However, PDESGD consistently obtains the best validation error. This result is almost always robust to the initialization since the

$\times 10^{-3}$	CH	D	BH	CCPP	ASN
SGD	14.73, 15.29	25.83, 26.99	5.38, 6.77	3.04, 3.06	5.39, 7.51
ADAM	15.02, 16.08	26.59, 28.02	5.58, **6.13**	3.05, 3.07	9.96, 12.01
PDESGD	**14.03**, **14.56**	**24.96**, **26.28**	**4.99**, 6.63	**2.85**, **2.94**	**3.25**, **6.36**

Table 5.2: **best, mean** validation error for the five considered UCI datasets.

mean of PDESGD is only beaten by ADAM once, on BH.

In summary, in this benchmark, PDESGD is at least better than SGD and ADAM, in terms of convergence speed and validation error. These results are all the more so encouraging since ADAM is a state-of-the-art, widely adopted optimization algorithm.

Remark. *For the Diabetes dataset, the validation error goes up after a few iterations. It is an illustration of the overfitting phenomenon. Still, in that case, it does not prevent PDESGD from obtaining the best validation error, for* ***best*** *and* ***mean****.*

5.5 Discussion

The work presented in this chapter is mainly theoretical. The experiments with PDESGD are convincing, but the test cases are too simplistic to conclude about their relevance for large-scale deep learning problems. In addition, the analysis only holds for $p_{out} = 1$. Unfortunately, we have not been able to study its application to high dimensional problems involving large neural networks in the time span of the thesis. Such a study would be in the continuity of this work: it is the closest perspective.

5.5.1 Perspectives

Even if the presented results focus on PDESGD, the PDE framework is also one of the main contributions of this work. There are many other things to explore beyond PDESGD:

- First, more experimental work could help to elaborate on the choices for α and $\boldsymbol{\theta}^* - \boldsymbol{\theta}$. To that end, it should be helpful to study the properties of the solved PDE for each different choice.
- The PDE framework relies on the assumption that $\hat{f} \sim 0$. There is an interest in assessing this assumption to improve the optimization possibly. The report of Poëtte et al. (2021) is an attempt to evaluate this hypothesis using transport PDE theory.
- The classical SGD could benefit from the PDE framework. Indeed, equation (5.16) provides a principled formula for noise injection based on the central limit theorem.

It could conciliate the advantages of large and small batch training. Indeed, a large batch could be used to estimate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ more accurately, and at the same time, a noise modeling the stochasticity of small batches could be explicitly injected. This approach has been tested, although not extensively studied in Section 5.4.4.2, and would deserve more thorough investigations. Moreover, controlled stochasticity could have other advantages, such as automating learning rate schedules or automatically stopping the training by principled criteria like statistical tests (similarly to Lang et al. (2019)). The learning rate limitation could also bring advantages to enforce the process's stability or control the accuracy of the obtained minimum, which could be of interest for the characterization of the different sources of error in Machine learning, as described in Section 2.4.1.

- As seen in the experiments of Section 5.4.3.3, the stability granted by the step size limitation could allow starting the optimization by an exploration step to find a good initialization. Then, reducing the step size would make the algorithm switch back to a more classical optimization setting. This approach has already been discussed by Ye et al. (2017) and could benefit from the PDE framework.

5.5.2 Interplay between numerical analysis and machine learning

This chapter focused on the optimization step of the supervised deep learning process. This step is particularly interesting regarding our final goal because it allows improving the error for no additional cost in the neural network run time. Finally, we achieved valuable results by revisiting the optimization at play in machine learning using stochastic PDE theory. The application range of the obtained findings is broader than numerical simulations and encompasses all optimization-based machine learning problems.

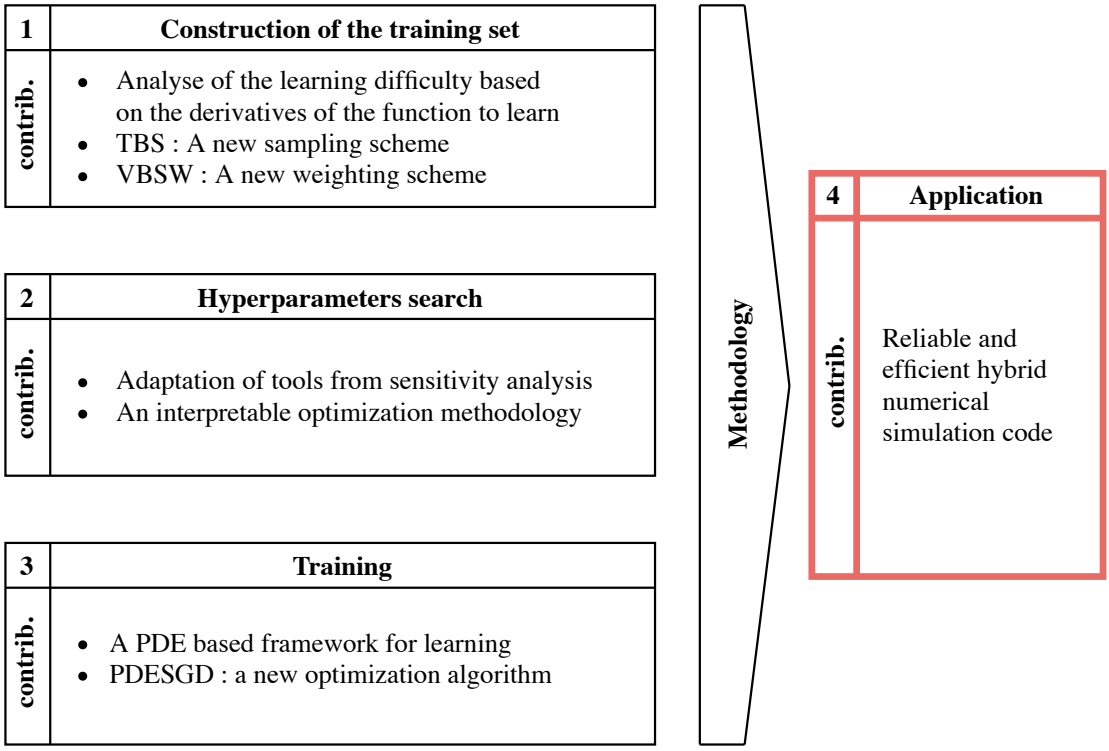
Chapter 6

Efficient hybrid numerical simulations (with guarantees)

In the previous chapters, we have investigated the three main steps of supervised learning in light of the concerns and specificities of numerical simulation and scientific computing. In this chapter, we apply our findings to a supervised learning task that aims at accelerating a multi-physics CFD simulation code.

We describe a neural network-based acceleration approach that is applicable to any numerical simulation codes involving the resolution of coupled systems of equations. This approach relies on the identification of a pattern in such codes that emphasizes the possibility to approximate a part of the code with a neural network. It yields a hybrid simulation code that leverages the implementation advantages and appealing performances of neural networks.

We apply this approach to the simulation of a hypersonic flow around an object during an atmospheric reentry. To accurately simulate this phenomenon, it is necessary to compute the chemical equilibrium of species found in the fluid. We approximate the chemical reactions with a neural network and obtain a significant speed-up - up to a factor 18.7 - for a comparable accuracy. Using error analysis based on uncertainty propagation, we show that the hybrid code's error is negligible as compared to other sources of errors that classical numerical simulation codes usually undergo. We also describe how to obtain the exact same predictions with the hybrid code, by using its predictions to initialize the computations of the original code.



Methodology for supervised deep learning in numerical simulations and contributions of the thesis

Contents

6.1 State of the art

Recently, the expansion of machine learning and its success in computer vision, natural language understanding, and pattern recognition has stimulated interest in its use in scientific computing. Some publications emphasize the need for whole scientific communities to integrate machine learning in their best practices, for instance, in numerical sciences (von Rueden et al., 2020); high energy physics (Albertsson et al., 2018); or molecular simulation (Noé et al., 2020).

Besides, machine learning and, more generally, statistical methods are already used to improve the performances of numerical simulations, with different degrees of modifications of simulation codes, that we detail in the following sections. First, machine learning can be used outside of the code and guide its execution or its conception. On the contrary, the code can be completely replaced by a machine learning model to enable parametric studies that could not be conducted with the original, too lengthy simulation code. This approach is called surrogate modeling. Finally, a third, intermediate degree consists of replacing a part of the simulation code with a machine learning model, yielding a hybrid code. Our approach falls under this last category, which aims at accelerating computer codes, but can also be used to guide the conception of these codes.

6.1.1 Helping numerical simulations from the outside

Machine learning can be considered a tool to improve numerical simulation codes. Some works leverage the ability of machine learning to process and analyze large volumes of data, experimental or not: data mining and machine learning-based data processing can be used to help simulations in different ways. Roscher et al. (2020) recommend the use of machine learning to extract patterns from data in order to help scientists understand a natural phenomenon and help for the conception of simulation codes. For instance, Reichstein et al. (2019) extract knowledge from geospatial data to enrich simulation codes. It is also used to improve agent-based simulations (Arroyo et al., 2010), aircraft fleet simulations (Painter et al., 2006), or to help simulation-based car conception (Bohn et al., 2013) by looking for more informative inputs. Similarly, Plattner et al. (2017); Zimmerman and Bowman (2015);

Doerr and De Fabritiis (2014) combine data-mining with active learning to improve molecular simulations. Finally, the authors of Lahoz et al. (2010); Baker et al. (2018) argue for the use of machine learning-based data processing to improve earth sciences and biological simulations.

Machine learning can also be used to improve the building blocks of existing simulation codes. At run-time, mesh adaptation Fidkowski and Chen (2021) can help to design efficient meshes, for instance, by detecting the shock in fluid dynamic (Beck et al., 2020). Furthermore, the parametrization of such codes can also be optimized using physics-informed neural networks (Raissi et al., 2017, 2019). These last techniques have become more and more popular lately, and we refer to Cai et al. (2021) for an exhaustive overview of its applications.

There are plenty of ways to improve numerical simulations using machine learning based data analysis. However, such methods do not explicitly and directly optimize the computational time of the code or one of its components. Therefore, all these works are complementary to ours.

6.1.2 Surrogate modeling: replacing the whole code for parametric studies

In several domains like physical sciences, economics, biology, or climatology, simulation codes are designed to predict the behavior of complex systems given input parameters. These input parameters can be made variable either to model uncertainty of their knowledge, to study the sensitivity of the prediction to these parameters, or to optimize the choice of these parameters in order to satisfy some practical design constraints (calibration). A parametric study is unavoidable in each case. However, such studies always imply repeated calls to the code with different input parameters value. When the code is computationally expensive, it may be unaffordable. The field of surrogate modeling has been developed in order to tackle this problem. The idea is to approximate the whole simulation code, seen as a black box, with a statistical model. This model is computationally cheaper than the original code, so it can be used in place of the code for parametric studies. Surrogate modeling can be performed with various machine learning models.

6.1.2.1 A variety of surrogate models

Gaussian Process regression, as popularized for machine learning by Rasmussen and Williams (2005) has percolated up to a leading position in surrogate modeling. Gaussian processes-based sensitivity analysis, introduced as Bayesian sensitivity analysis in Oakley and O'Hagan (2004) models the output of the code with a Gaussian process and computes the sensitivity indices of input parameters using this approximation. See Da Veiga et al. (2009) or Marrel et al. (2015) for examples of recent improvements of this approach. For calibration, Kennedy and O'Hagan (2001) introduced a Bayesian framework for calibration of computer models, which remains the top canonical approach (Gramacy, 2020). The remaining field based on gaussian process

surrogates models is called Bayesian optimization. It has already been used in Chapter 4, which illustrates its range. The review of Shahriari et al. (2016) gives an idea of the diversity of its application. Bayesian optimization is also popular in numerical simulations for engineering, since the works of Jones et al. (1998a) and the continuous work of Santner et al. (2018).

Based on the seminal work of Wiener (1938), Polynomial chaos (Xiu and Karniadakis, 2002; Blatman and Sudret, 2008) is also very popular, especially in problems of uncertainty quantification and sensitivity analysis Sudret (2008); Lucor et al. (2007c). Since the work of Le Maître and Knio (2004), it is extensively used in non intrusive uncertainty propagation (Hosder et al., 2006; D. et al., 2003; Poëtte et al., 2009), but also for calibration Lucor et al. (2007b); Congedo et al. (2011).

First applications of neural networks, e.g. to assess e probability of failure (Hurtado, 2001), or structural reliability (Papadrakakis et al., 1996) dates back to the late 1990s. However, their use for surrogate modeling has become more widespread in the last decade due to the recent breakthroughs of neural networks as successful statistical models. For uncertainty quantification, neural networks are often considered as classical surrogates (Zhu et al., 2019; Winovich et al., 2019). Nonetheless, some works develop and use Bayesian approaches that allow naturally propagating uncertainty similarly to Gaussian processes (Gal, 2016; Lampinen and Vehtari, 2001; Kwon et al., 2020). Neural networks are also used as surrogate models for calibration and optimization (Koziel and Leifsson, 2012; Guo et al., 2016; Rudy et al., 2017), or simply to replace computer codes (Feng et al., 2018; Mao et al., 2021; Lu et al., 2021).

Although they are less common, other surrogate models such as random forest (Breiman, 2001), gradient boosting machine (Friedman, 2000) or Multivariate Adaptive Regression Splines (Friedman, 1991) are also used in similar ways, as described and tested in Storlie and Helton (2008) and Storlie et al. (2009).

6.1.2.2 Limits of surrogate modeling

Surrogate modeling is quite similar to our approach since we approximate a - part of a - computer code with a neural network. However, contrarily to us, it considers the simulation code as a black box.

Its strength relies on the cost efficiency of the surrogate as compared to the approximated code. However, for high dimensional simulation codes that take days to return a prediction, surrogate modeling can become irrelevant because of the lack of data and its high dimensionality. Moreover, even in workable cases, the accuracy of surrogate models as predictors is often limited due to the computational cost of obtaining training data points.

In this work, we open the black box to plug a surrogate model inside the original code, leading to a hybrid simulation code. The part of the code that the neural network replaces is not so costly when executed as a standalone application: it allows constructing a large training database, thereby dramatically improving the prediction performances of the surrogate. In

addition, we leverage the ability of neural networks to process data in batches very efficiently - thanks to the matrix product structure of their implementation - to improve the performances of the hybrid code. As a result, there are two points to insist on :

- The approach of approximating a part of a simulation code allows to speed-up computer code so expensive, in terms of computational cost and dimensionality, that classical surrogate modeling would fail.
- Embedding the neural network inside the simulation code and generating a large training database yields a hybrid code that returns predictions whose accuracy is comparable to that of the original code. It enables its use in contexts where prediction accuracy must be ensured.

6.1.3 Hybridization of machine learning and numerical simulations

The idea of constructing a hybrid code based on both numerical simulation and machine learning has already been explored in previous works. Most of them replace a part of a simulation code with a machine learning model. In molecular simulations, Behler and Parrinello (2007) use a neural network approximation of potential energy surface, and Stecher et al. (2014) use Gaussian processes to sample Gibbs free energy surface, opening the avenue for applications based on such methodologies Schneider et al. (2017); Mones et al. (2016); Bereau et al. (2018); Brockherde et al. (2017). In CFD simulations, (Danvin et al., 2021) use neural networks to predict mode amplification and (Bourriaud et al., 2020) to prevent troubled cells and adapt reconstruction polynomial degrees. Other works are even closer to our approach. Han et al. (2019) use neural networks to approximate physical components of multi-physics problems for electro-thermal simulation when conducting electrosurgery. In Kluth et al. (2020); Kluth et al. (2019), the authors approximate non-local thermodynamic equilibrium in the simulation of inertial confinement fusion. Moreover, (Danvin et al., 2021) use neural networks to predict mode amplification in CFD simulations.

In our work, we also initialize a numerical simulation code with the prediction of a hybrid code. This methodology is similar to these of Huang et al. (2020), where the prediction of a neural network is given as initialization to a PDE-based solver. However, unlike in our work, a new neural network is trained for each test case. Moreover, the neural network is a surrogate model trained to approximate the whole code. Hence, it suffers from the difficulties of surrogate models mentioned in the previous paragraph.

This work aims at strengthening the interest of the scientific computing community in following such hybrid approaches. Indeed, we give a general framework that emphasizes the applicability of this approach to a large range of multi-component simulation settings. We demonstrate its potential gains in computational time with limited accuracy loss in a multi-physics CFD code. In addition, we exploit a specificity of neural networks that previous works do not leverage. Neural networks implementation is very efficient and boils down to a sequence of matrix multiplications, which allows for efficient processing of batch, arrays-like inputs.

This characteristic is particularly suited to numerical simulations, often conducted on meshes which are arrays-like structures. When replacing physical components that can not naturally be executed in parallel, the perspectives of computational gains are even larger.

6.2 A general approach for constructing of hybrid simulation codes

Numerical simulation codes often share a common structure based on the coupling of several equations. This section describes the pattern behind this structure and justifies why and how neural network-based hybrid simulation codes can be constructed by leveraging this pattern.

6.2.1 Structure of simulation codes based on coupled equations

A multi-physics simulation code often solves a system of several components that model different physics. In this part, we formalize the mathematical framework of this system. To simplify the framework, we only consider a code with two coupled systems of equations. The system can be written :

$$\begin{cases} F_1(\boldsymbol{\eta}, \boldsymbol{u}, \boldsymbol{\alpha}) = 0, & \text{(6.1a)} \\ F_2(\boldsymbol{u}, \boldsymbol{\eta}, \boldsymbol{\alpha}) = 0, & \text{(6.1b)} \end{cases}$$

where

- $\boldsymbol{u}$ and $\boldsymbol{\eta}$ are vectors of unknowns,
- $\boldsymbol{\alpha}$ is a vector of physical parameters, that are not computed during the simulation (e.g. physical or chemical constants),
- F_1 and F_2 are mathematical (possibly differential) operators.

Such a formalism encompasses physical simulations like non-equilibrium thermodynamics, neutronic, combustion, detonic... Examples of such systems are illustrated in the following.

Example 1 : Boltzman-Bateman equations in neutronic

Some neutronic applications (Bernède and Poëtte, 2018; Dufek et al., 2013) consist in solving the system :

$$\begin{cases} \partial_t \boldsymbol{u} + v\omega\nabla\boldsymbol{u} + v\Sigma_t\eta\boldsymbol{u} - v\Sigma_s\boldsymbol{\eta}\displaystyle\int P_s\boldsymbol{u}d\omega' = 0 \\ \partial_t\boldsymbol{\eta} - \boldsymbol{\eta}\Sigma_r v\displaystyle\int \boldsymbol{u}d\omega = 0. \end{cases}$$

- $\boldsymbol{u}$ is the particle density, $\boldsymbol{\eta}$ the material density
- $\boldsymbol{\alpha} = (\Sigma_t, \Sigma_r, \Sigma_s)$ contains the cross sections, fixed parameters of the simulation, and P_s is a scattering law.
- F_1 and F_2 are respectively
  - The Boltzmann equation to model the transport of particles in matter (costly to solve)
  - The Bateman equations to model the reactions of particles with matter (cheap to solve independently, but costly when embedded)

Example 2 : Navier-Stockes-Bateman equations in CFD

Some CFD applications (Day and Bell, 2000; Najm et al., 1998) consist in solving the system :

$$\begin{cases} \partial_t\rho\boldsymbol{v} + \nabla(\rho\boldsymbol{v}\otimes\boldsymbol{v} + p(\boldsymbol{\eta})) - \boldsymbol{\mu}\nabla^2(\boldsymbol{v}) - \dfrac{1}{3}\boldsymbol{\mu}\nabla(\nabla\boldsymbol{v}) + \rho(\boldsymbol{\eta})\boldsymbol{g} = 0 \\ \partial_t\boldsymbol{\eta} - \Sigma_r\boldsymbol{\eta}\cdot\boldsymbol{\eta} = 0, \end{cases}$$

where $\otimes$ is the outer product.

- $\boldsymbol{u} = (\boldsymbol{v}, \rho, p)$ contains the flow velocity, the density and the pressure, and $\boldsymbol{\eta}$ is a vector of densities of molecules.
- $\boldsymbol{\alpha} = (\mu, \boldsymbol{g}, \Sigma_r)$ contains the dynamic viscosity, the gravitational constant and a reaction constants, fixed parameters of the simulation.
- F_1 and F_2 are respectively
  - The convective form of the Navier-Stokes momentum equation to model the flow of a compressible fluid (costly to solve)
  - The Bateman equations to model reactions within the components of the fluid (cheap to solve independently, but costly when embedded)

The resolutions of such codes often share a common pattern. In order to solve equation (6.1a),

the solver needs to solve equation (6.1b). As a result, equation (6.1a) is costly to solve because it regularly calls for the resolution of equation (6.1b). Indeed, most of the computational time is spent on these calls. All this justifies all the more the construction of a hybrid model based on the approximation of the resolution of the second equation.

6.2.2 Neural networks as reduced models

Using approximations to avoid the computational time induced by the resolution of equation (6.1b) is already in practice in the field of numerical simulations. For instance, some abacuses are constructed using the solver of equation (6.1b), and this solver is replaced by interpolation in these abacuses at run-time (Sigrist, 2019, 2020). However, this method becomes vacuous when the dimension increases because the number of points needed to obtain a proper interpolation increases, whereas the complexity of a search depends on the number of points in the abacus. Another example is the use of simplifying hypotheses to make the resolution of equation (6.1b) cheaper. These hypotheses may strongly accelerate the computation, but the simulated phenomenon may be very different from the one we are interested in. This last point is illustrated in Section 6.4.

Instead of using such traditional reduced models, we use neural networks as surrogate models for the resolution of equation (6.1b). Neural networks stand out from traditional surrogate models such as chaos polynomials or Gaussian processes because :

- The complexity of a forward pass to obtain a prediction scales linearly with the dimension of the problem (unlike for chaos polynomials), and the execution time does not depend on the number of points used for the training (unlike Gaussian processes). In other words, they are particularly resistant to the curse of dimensionality. Hence, we can construct a large database using the solver of equation (6.1b), which is likely to foster approximation accuracy.
- The recent advances in the research field of AI have stimulated many breakthroughs (dropout (Srivastava et al., 2014), batch-normalization (Ioffe and Szegedy, 2015), residual connections (He et al., 2015), etc...) that make neural networks more and more accurate.
- The implementation of neural networks using TensorFlow C api makes it suited to the integration inside an already existing high performances simulation code. Besides, since a prediction boils down to a succession of matrix products, it is very cost-effective when called on arrays-like structures, like meshes.

6.3 Test case: a CFD code coupled with chemical equilibrium

We consider a multi-physics CFD code, which simulates the dynamic of a multi-atomic gas fluid around an object of a given shape. The gas stream meets the object at hypersonic speed, creating a shock. The temperature and pressure increase inside the shock so that chemical reactions can occur between the chemical species that are found in the fluid. The phenomenon is illustrated in Figure 6.1. The code is executable on simple test cases but can be scaled up to run computationally demanding simulations, which makes it perfectly suited to our study.

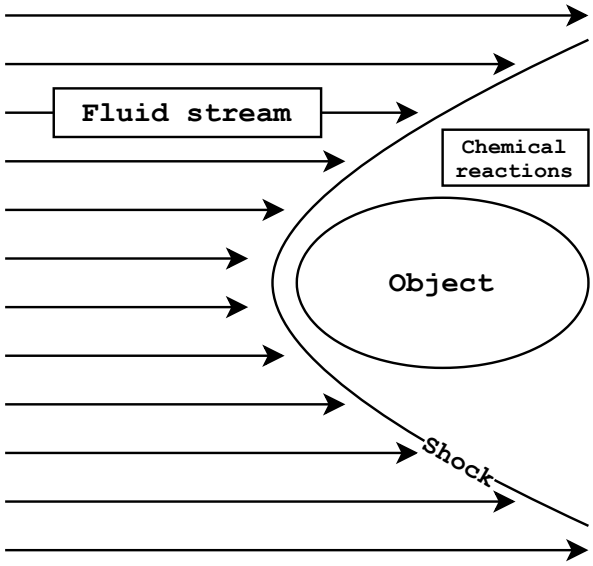


Figure 6.1: Illustration of the simulation.

This code can be used, for instance, to simulate the entry of a space shuttle into the earth's atmosphere. In that case, the object is a space shuttle, and the fluid is the air: a gas with two atomic elements, oxygen O and azote N. We are interested in the pressure and temperature at the boundary of the shuttle, which allows predicting whether the shuttle will be destroyed or not in these conditions.

6.3.1 Euler equation coupled with Gibbs free energy minimization

The fluid dynamic is simulated by the resolution of Euler equations:

$$\partial_t U + \nabla F(U) = 0, \tag{6.2}$$

with

$$U = \begin{pmatrix} \rho_1 \\ \dots \\ \rho_{n_e} \\ \rho v \\ \rho w \\ \rho E \end{pmatrix}, F_x(U) = \begin{pmatrix} \rho_1 v \\ \dots \\ \rho_{n_e} v \\ \rho v^2 + p \\ \rho v w \\ \rho u(E + \frac{p}{\rho}) \end{pmatrix}, F_y(U) = \begin{pmatrix} \rho_1 w \\ \dots \\ \rho_{n_e} w \\ \rho v w \\ \rho w^2 + p \\ \rho w(E + \frac{p}{\rho}) \end{pmatrix},$$

where $\rho_i = \rho\tilde{\eta}_i$ is the partial density of element $i \in \{1, ..., n_e\}$ (for n_e elements - in the case of the atmosphere, $n_e = 2$ and the elements are O and N), $\rho = \sum_{i=1}^{n_e} \rho_i$ is the density of the fluid, v and w are respectively the horizontal and vertical speeds, $E = \epsilon + \frac{v^2+w^2}{2}$ is the total energy and p is the pressure. This system of equations is not closed and we need another equation to obtain the pressure.

A first solution is to make the hypothesis that the fluid is a perfect gas, in which case we have :

$$p = (\gamma - 1)\rho\epsilon,$$

where γ is the Laplace constant. In the case of a diatomic gas, $\gamma = 1.4$. However, the hypothesis of perfect gases is coarse, and this value of γ holds only for diatomic gases. In a more general case, the pressure p can be accurately computed by simulating the chemical equilibrium between the species produced by the chemical reactions that occur during the dynamic. For instance, we could consider the reaction

$$N + O \leftrightharpoons NO.$$

In that case, there are two elements, N and O, and there are three species, N, O, and NO. Let us consider n_s species. Then the pressure is obtained using

$$\begin{cases} p(\rho, \epsilon, \boldsymbol{\eta}) = \dfrac{\rho R T(\epsilon, \boldsymbol{\eta})}{m} \\ T(\epsilon, \boldsymbol{\eta}) = \dfrac{\epsilon - \sum_{i=1}^{n_s} \eta_i h_i^0}{\sum_{i=1}^{n_s} \eta_i C_{vi}}, \end{cases} \tag{6.3}$$

where R is the universal gas constant, h_i^0 and C_{vi} are respectively the mass enthalpy of formation and the mass heat capacity at constant volume of specie i, and $m = \sum_{i=1}^{n_s} x_i m_i$, with m_i and x_i the molar mass and the molar fraction of the specie i. All these quantities are associated with the considered species. These expressions also involve $\boldsymbol{\eta} = (\eta_1, ..., \eta_{n_s})$, where η_i is the mass fraction of specie i. In order to obtain $\boldsymbol{\eta}$, the CFD code uses a method based on the minimization of the Gibbs free energy G, obtained by solving

$$\Delta G(\boldsymbol{\eta}, U, \boldsymbol{\alpha}_G) = 0, \tag{6.4}$$

where $\boldsymbol{\alpha}_G$ contains all the parameters used in the minimization, which are the numbers of moles and the Gibbs functions of pure species. This method is implemented by the library Mutation++ (Scoggins et al., 2020).

Finally, we can link this code to the general structure of Section 6.2.1. The code writes:

$$\begin{cases} \partial_t U(\boldsymbol{\eta}) + \nabla F(U(\boldsymbol{\eta})) = 0, \\ p(\rho, \epsilon, \boldsymbol{\eta}) = \dfrac{\rho R T(\epsilon, \boldsymbol{\eta})}{m}, \\ T(\epsilon, \boldsymbol{\eta}) = \dfrac{\epsilon - \sum_{i=1}^{n_s} \eta_i h_i^0}{\sum_{i=1}^{n_s} \eta_i C_{vi}}, \\ \Delta G(\boldsymbol{\eta}, U, \boldsymbol{\alpha}_G) = 0 \end{cases}$$

- The vector $\boldsymbol{u} = (\rho_{i|_{i=1,n_e}}, v, w, \epsilon, p, T)$ and the vector $\boldsymbol{\eta} = (\eta_1, ..., \eta_{n_s})$.
- $\boldsymbol{\alpha} = (R, h^0_{i|_{i=1,n_s}}, C_{v_i|_{i=1,n_s}}, \boldsymbol{\alpha}_G)$.
- F_1 and F_2 are respectively
  - The operator concatenating equation (6.2) and equation (6.3).
  - The equation of minimization of the Gibbs free energy, equation (6.4).

6.3.2 Mutation++

In the CFD code, Mutation++ takes as inputs ρ, the density, ϵ, the mixture energy, and the mole fractions of the elements initially found in the fluid. It outputs $\{\eta_1, ..., \eta_{n_s}\}$ the mass fractions of the mixture of n_s chemical species but also additional metrics: c, the speed of sound, C_p the heat at constant pressure, C_v, the heat at constant volume, p the pressure and T the temperature after the reactions.

In practice, the mass fractions of the input elements are always the same since the fluid is permanently renewed by the dynamic. Hence, for the same test case, we only consider 2 inputs: ρ and ϵ. The function of Mutation ++ can be summarized as :

$$\mathbf{M}_{++} \quad : \quad \begin{pmatrix} \rho \\ \epsilon \end{pmatrix} \in \mathbb{R}^2 \quad \longrightarrow \quad \begin{pmatrix} \eta_1 \\ ... \\ \eta_{n_s} \\ P \\ T \\ C_p \\ C_v \\ c \end{pmatrix}, \in \mathbb{R}^{n_s+5} \tag{6.5}$$

with $\mathbf{M}_{++}$ denoting Mutation++. In the following, we aim at constructing a neural network approximating $\mathbf{M}_{++}$.

6.3.3 The computational burden of chemical equilibrium

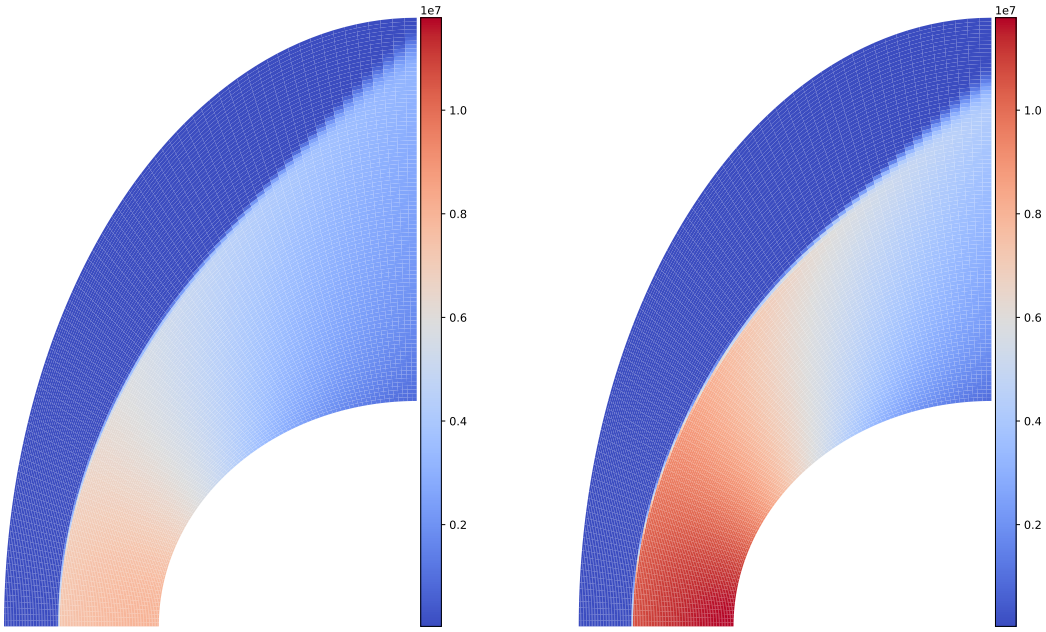
The code solving F_1 is executed on a mesh using a resolution scheme detailed in Peluchon (2017). When executed alone, i.e. using the perfect gas hypothesis, the resolution is cost-effective. However, when considering chemical equilibrium, the minimization of the Gibbs free energy using Mutation++ has to be conducted in each cell of the mesh for each time step of the resolution. In that case, the whole resolution is more accurate but becomes expensive.

To illustrate this slow-down, we simulate the dynamic for a sphere entering the atmosphere at a speed of 4930 $m.s^{-1}$ (Mach 16) on a structured mesh, and consider one reaction between Oxygen (O) and nitrogen (N): $O + N \rightarrow ON$. We call this code MPP. We compare the prediction of MPP with that of the simulation of a perfect gas without reactions, which we call PG. The parameters of the simulation are gathered in Table 6.1.

Input	value
Elements (`elem:fraction`)	O:0.2, N:0.8
Upstream pressure	35737.40Pa
Upstream temperature	216.57K
Upstream speed	4930.83 $m.s^{-1}$
Chemical species	N, O, NO

Table 6.1: Simulation parameters and boundary conditions for the toy example

Figure 6.2 plots the pressure field of the fluid around the object for MPP and PG. The pressure maps clearly differ, emphasizing the importance of simulating finer physics (here, chemical equilibrium). However, the cost of the code is significantly affected: the execution time of MPP is 4090 seconds, against 81 for PG, which is 56 times higher.



(a) With chemical equilibrium - 4090 seconds (b) Without chemical equilibrium - 81 seconds

Figure 6.2: Pressure field for the code with and without chemical equilibrium.

Simulating chemical equilibrium takes most of the computational time of the code, but on the other hand, Figure 6.2 shows that we cannot avoid simulating it to obtain accurate predictions. There are consequently high stakes at accelerating the computations related to the chemistry of the problem.

6.4 Acceleration of the simulation code

This sections aims at demonstrating the benefits of hybrid codes, but also to showcase the previous contributions of the thesis. We consider three different types of codes:

- PG: Execution of the CFD code without simulating any chemical reactions (perfect gas).
- MPP: Execution of the CFD code with the simulation of chemical reactions using Mutation++.

- NN: Execution of the CFD code with the simulation of chemical reactions using a neural network approximating Mutation++.

For each code, we perform the computations on a low-resolution mesh (30×100) and on a high-resolution mesh (90×100). For a code C, we denote respectively by C_high and by C_low its execution on the high and low-resolution mesh. We execute the codes on two different mesh resolutions for two reasons. First, the prediction of MPP_high can be taken as a reference to compute the prediction errors of the other codes, including MPP_low. Second, as we shall see, the high-resolution version of NN is still faster than the low resolution of MPP while being more accurate for some output metrics (see Section 6.4.2.2).

6.4.1 Methodology for designing NN hybrid code

In order to construct NN hybrid code, the first step is to design and train the neural network that will be embedded inside the code. This process follows the three steps described in 2, and around which the previous chapters were organized: construct the training database; select hyperparameters; and train the network. In our case, and following remark 2.5, we include the training database construction and the training in the hyperparameter search, since these steps may have their own hyperparameters.

To demonstrate the benefits of hybrid codes and to illustrate the advantages of the previous contributions, we compare two methodologies: the first, **vanilla**, and the second, **thesis**, characterized as:

vanilla

- Construct the database with uniform sampling.
- Find hyperparameters using a random search.

thesis

- Allows to construct the database using variance based sampling or to weight it using variance based sam-Variple weighting (3).
- Find hyperparameters (including for VBSW) using TS-GPBO (4.5).

ance based sampling refers to the same algorithm as TBS 1, but with $Df_{\boldsymbol{\epsilon}}^2$ estimated using local variance, like in equation (3.4). The input space is defined by $\rho \in [0.1, 3.8]$ and $\epsilon \in [2.07503 \times 10^7, 3 \times 10^8]$. For ϵ, the zero is not the same for the CFD code as for $\mathbf{M}_{++}$, so its boundaries are chosen so that the lower bound of ϵ matches the zero of the CFD code. In addition, its distribution is log uniform. For ρ, the boundaries are chosen wide enough to encompass all the possible values of the test case.

The neural network that we use is a Multi Layer Perceptron, and the hyperparameter space is described in **Appendix B**. It is trained using Tensorflow in python on a training dataset of 170000 points, and the hyperparameters are selected using a validation data set of 20000

points. Then it is exported and embedded into the original code in place of Mutation++ using the Tensorflow C api with a wrapper, CppFlow[1].

As opposed to Mutation++, the calls of the C++ implementation of a neural network are naturally vectorized. Therefore, we also modify the original code to leverage the implementation advantages of neural networks so that it can be called on the whole mesh, in a batch fashion.

Remark. *It turns out that this characteristic of neural networks is key to obtain a speed-up in our case. Indeed, when it is called on one input point, Mutation++ is faster than the neural network. The latter actually shines when it is called on the whole mesh, whereas Mutation++ has to be called sequentially on each cell.*

6.4.2 Application of the vanilla methodology

In this section, we assess the results of **vanilla**. We first study the predictions of the neural network for the approximation of Mutation++. Then, we introduce the first results obtained with NN, the hybrid code.

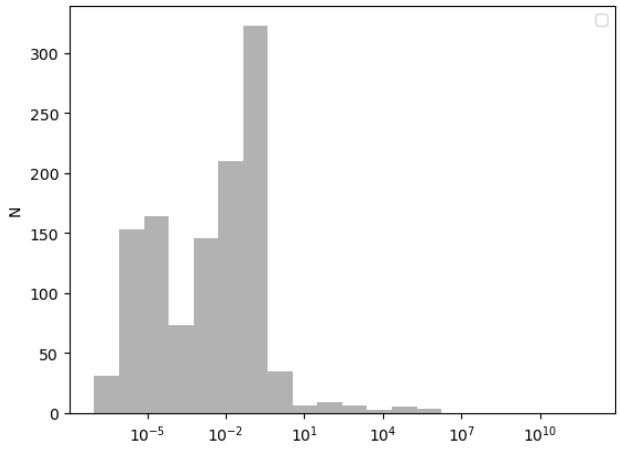
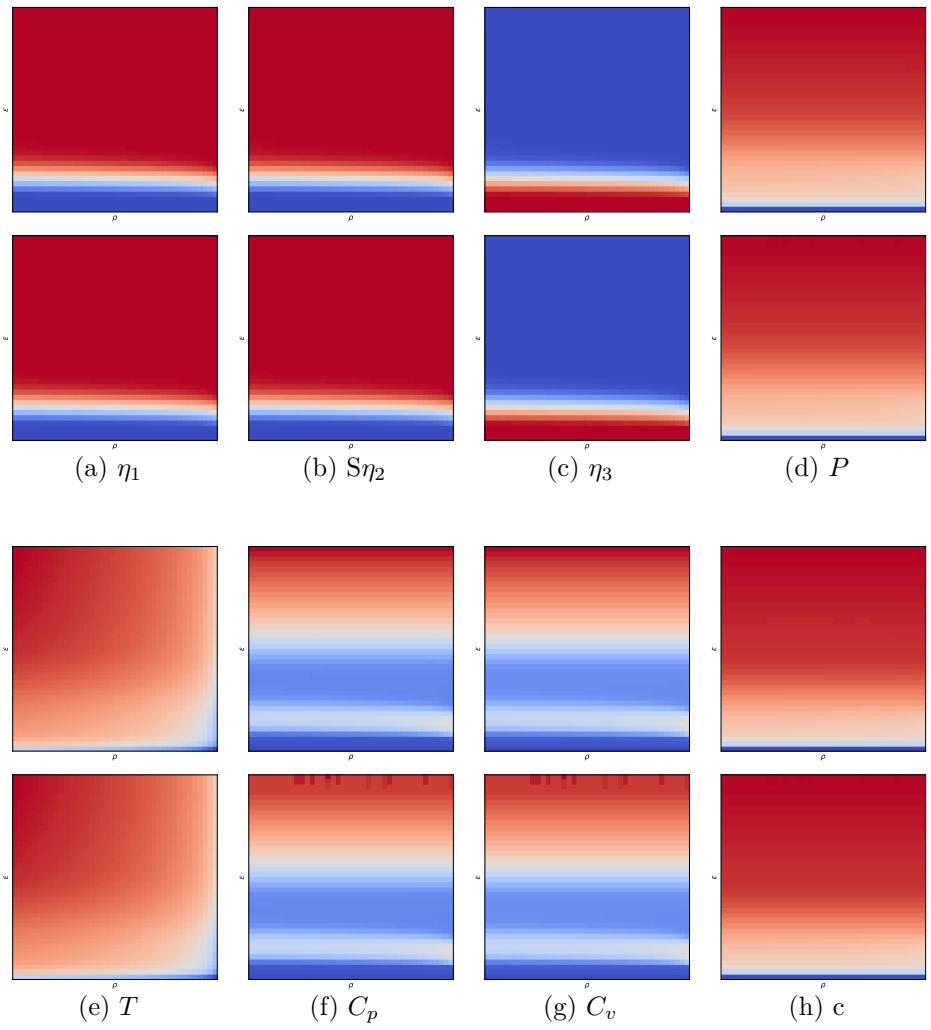


Figure 6.3: Histogram of the normalized L_2 validation error of the neural networks for approximating $\mathbf{M}_{++}$.

6.4.2.1 Approximation of Mutation++

We assess the performances of neural networks for approximating the toy case of $\mathbf{M}_{++}$ through an initial random search. The random search is based on 1000 different hyperparameter configurations uniformly sampled in the hyperparameter space. This space is described in **Appendix B**. Figure 6.3 plots the histogram of the errors obtained after this random search.

[1] `https://github.com/serizba/cppflow`



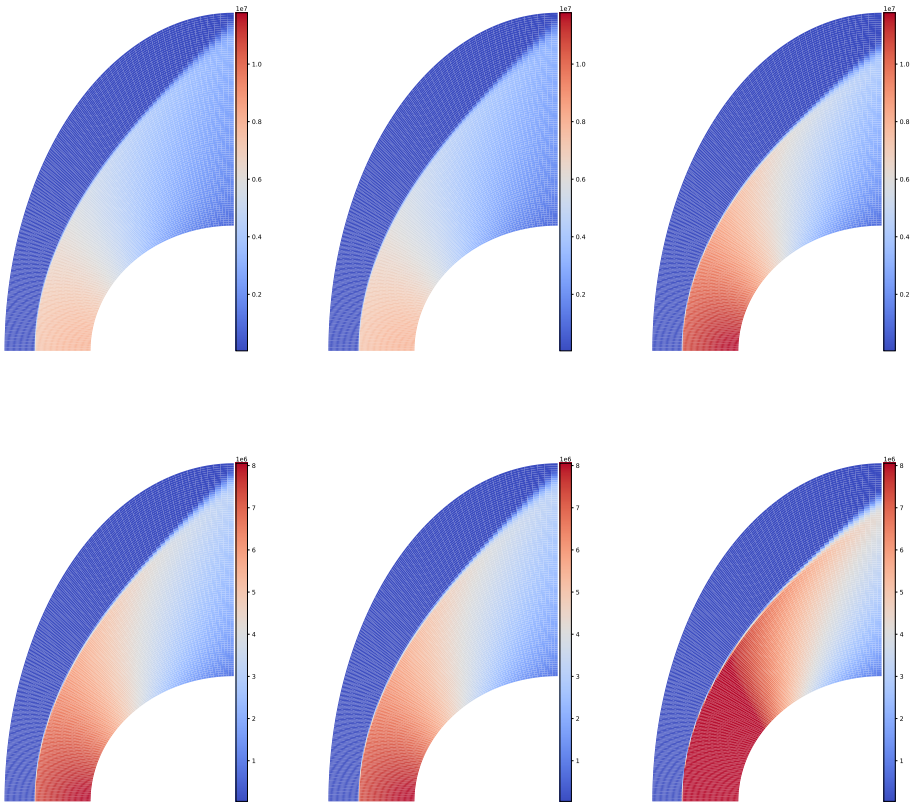
(a) η_1 (b) Sη_2 (c) η_3 (d) P

(e) T (f) C_p (g) C_v (h) c

Figure 6.4: Predictions of the neural network (top) and predictions of $\mathbf{M}_{++}$ (bottom).

This histogram once again demonstrates the impact of hyperparameters on neural networks performances: the best error (normalized L_2) is 9.37×10^{-8}, but it goes up to orders of 10^{10}. The neural network that yield the best error has `n_layers` $= 9$ and `n_units` $= 191$ units, which is close to the upper boundary of the search space for these hyperparameters.

We keep this neural network for our first experiments and provide a visualization of its predictions against the input space of $\mathbf{M}_{++}$ in Figure 6.4. For each output physical observable, the predictions are compared to the ground truth values computed by $\mathbf{M}_{++}$. No differences can be seen visually, which is a good point. Note that for C_p and C_v, there are artifacts in the predictions of $\mathbf{M}_{++}$ at the top of the domain, while there is none for the neural network. This may come from numerical instabilities of $\mathbf{M}_{++}$ that may disturb the computation. The neural network does not seem to be subject to such instabilities. This point is also illustrated in section 6.5. In the next section, we assess how these predictions are reflected in the hybrid code.



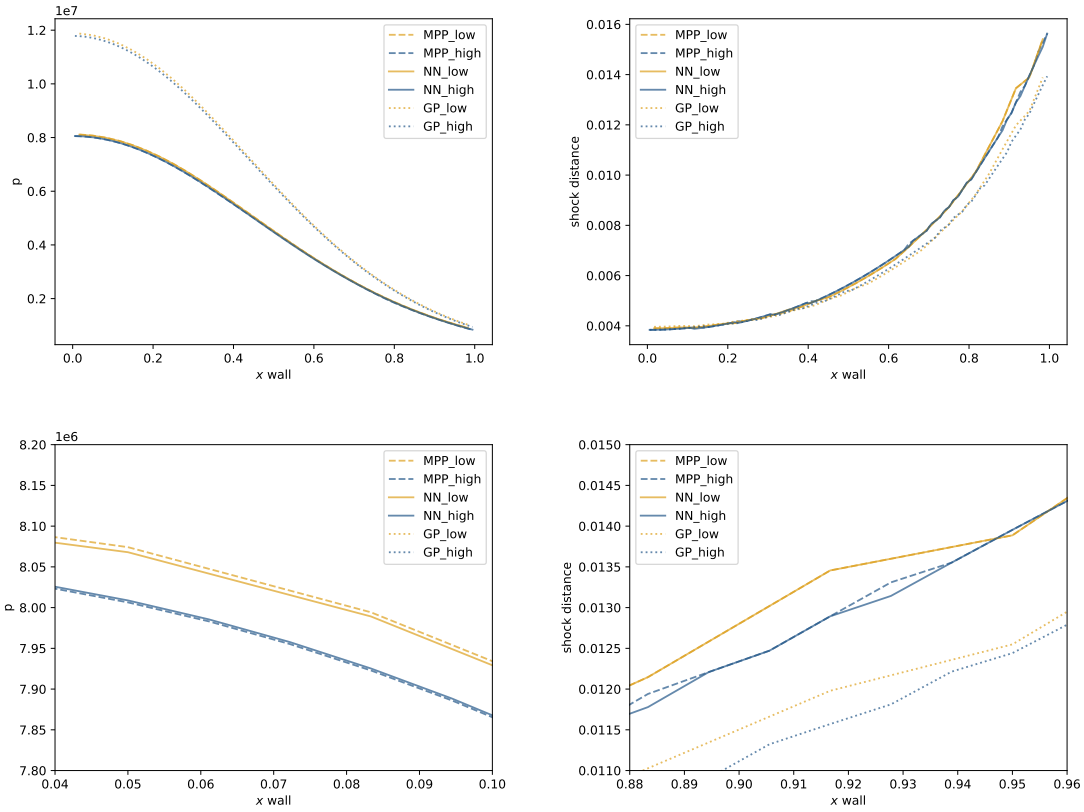
(a) MPP (b) NN (c) PG

Figure 6.5: Pressure field for MPP_high, NN_high and PG_high, with the scale defined by the minimum/maximum pressure values obtained by PG (top) and MPP (bottom).

6.4.2.2 Accuracy and performances of the hybrid code

After a first run of the three codes, MPP, NN, and PG, with different resolutions, we assess the predictions on the pressure. Figure 6.5 gathers the pressure field for the three codes executed in the high-resolution mesh, with different scales constructed from the minimum and maximum pressure values obtained by GP and MPP. Figure 6.6 gathers the prediction of the codes for the pressure profile and the shock distance projected on the wall of the object. The L_2 and L_∞ normalized errors for these curves are gathered in Tables 6.2 and 6.3. The tables also display the execution times of the different codes.

On Figure 6.5 and 6.6, it is visually clear that the PG code yields erroneous predictions. This shift justifies taking the chemistry of the phenomenon into account for this simulation. The origin of this error, namely the model error, is mentioned in Section 6.5. Furthermore, no visual differences can be detected between MPP and NN, which is a good point. To assess the prediction error of NN, we have to look at more quantitative results.



(a) Pressure profile

(b) Shock distance

Figure 6.6: Pressure profile and shock distance projected on the wall of the object. In the bottom line, zoom of the curves in the highest error area.

Table 6.2 displays the errors of NN and PG when compared to MPP for each resolution. They emphasize that the error of NN in each resolution is far lower than that of PG, for a significantly reduced computational time with respect to MPP. The time reduction factor is approximately the same for each resolution.

In Table 6.3, we compile the previous results, but we evaluate the L_2 and L_∞ errors and the time improvement factor with respect to MPP_high only. This comparison exhibits the error of MPP_low due to the lower resolution and enables its comparison with the error of both NN codes. Several points can be made regarding these results. The errors obtained with both NN codes are close to those of MPP_low. For the shock distance, the estimation obtained with NN_high is even better than with MPP_low. For the PG code, the prediction error is at least one decade higher than NN and MPP, regardless of the resolution. However, NN is closer to PG in terms of execution time, with 531 and 1209 for NN and 81 and 211 for PG, against 4090 and 9086 for MPP. To sum up, on the one hand, MPP and NN are comparable in terms of error, and on the other hand, NN is almost one decade faster than MPP. These results are comforting and motivate us to go further in the study of the method.

	MPP_low (ref)	NN_low	PG_low
Time (s)	4090	531	81
Impr. (×)	-	8.9	58.5
Pressure			
L^2	-	6.06×10^{-7}	7.74×10^{-2}
L^∞	-	1.13×10^{-3}	4.64×10^{-1}
Shock dist.			
L^2	-	7.85×10^{-7}	1.70×10^{-3}
L^∞	-	4.85×10^{-3}	9.89×10^{-2}

	MPP_high (ref)	NN_high	PG_high
Time (s)	9478	1209	211
Impr. (×)	-	7.8	44.9
Pressure			
L^2	-	1.20×10^{-6}	7.65×10^{-2}
L^∞	-	2.00×10^{-3}	4.62×10^{-1}
Shock dist.			
L^2	-	8.54×10^{-6}	1.65×10^{-3}
L^∞	-	1.06×10^{-2}	1.08×10^{-1}

Table 6.2: Execution times, with improvement factor with respect to the ref, and normalized L_2 and L_∞ errors for the different codes in the low and high resolution meshes.

	MPP_high (ref)	MPP_low	NN_high	NN_low	PG_high	PG_low
Time (s)	9478	4090	1209	531	211	81
Impr. (×)	-	2.3	7.8	17.9	44.9	117
Pressure						
L^2	-	3.19×10^{-5}	$\mathbf{1.20 \times 10^{-6}}$	3.23×10^{-5}	7.65×10^{-2}	8.15×10^{-2}
L^∞	-	9.07×10^{-3}	$\mathbf{2.00 \times 10^{-3}}$	9.35×10^{-3}	4.62×10^{-1}	4.75×10^{-1}
Shock dist.						
L^2	-	9.14×10^{-5}	$\mathbf{8.54 \times 10^{-6}}$	9.18×10^{-5}	1.65×10^{-3}	1.29×10^{-3}
L^∞	-	3.60×10^{-2}	$\mathbf{1.06 \times 10^{-2}}$	3.60×10^{-2}	1.08×10^{-1}	9.00×10^{-2}

Table 6.3: Execution times and normalized errors for the different codes.

6.4.3 Effect of HSIC analysis and Variance based sampling

In this section, we study the **thesis** methodology, i.e. the effect of HSIC based sensitivity analysis and variance-based sampling/weighting on the results of neural network and on the hybrid code NN.

To that end, we include a new categorical hyperparameter that we call `local_variance` $\in$ {`vbs`, `vbsw`, `None`}, where `vbs`, `vbsw` and `None` respectively denote the cases where we sample from the local variance, we weight the data using VBSW and we do not perform any sampling or weighting. When `local_variance` $\in$ {`vbs`, `vbsw`}, two conditional hyperparameters are involved: k, the number of points used to estimate the local variance, and m, the ratio used to construct the new distribution. The value `vbs` also comes with another hyperparameter : `sample_split` $\in [0, 1]$, which is the percentage of points to replace by samples of the new distribution.

6.4.3.1 Effect on the neural network

In order to perform the sensitivity analysis, an initial random search has to be conducted in order to evaluate the HSIC of each hyperparameter. Since 1000 points have already been sampled without `vbs` or `vbsw`, we sample 1000 additional points for each new value of `local_variance`. As a result, 3000 points are used to assess the HSIC. Their estimation for the main hyperparameters is illustrated in Figure 6.7.

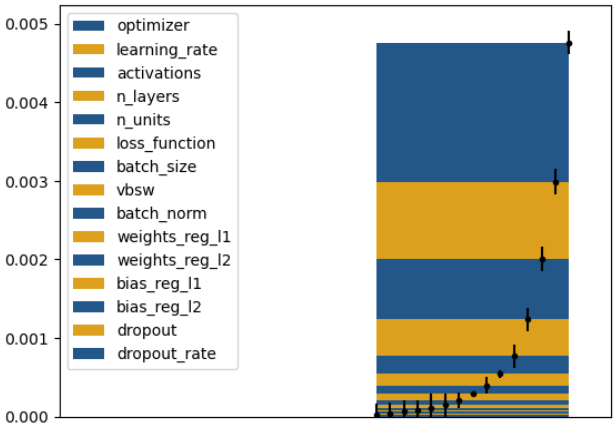


Figure 6.7: HSIC of the hyperparameters for L_2 error.

After this initial random search, the best neural network obtained has `n_layers` $= 8$ and `n_units` $= 193$, is trained with VBSW and reaches a validation L_2 error of 1.13×10^{-8}. It is almost one decade better than the previous best error, which illustrates the benefits of VBSW. The hyperparameters `optimizer`, `learning_rate`, `activations`, `n_layers`, `n_units` and `loss_function` seems to be the most impactive.

We carry on with the application of our previous work and apply the TS-GPBO methodology (see Chapter 4, Section 4.5). The obtained neural network reaches a L_2 error of 8.48×10^{-8}, with `n_layers` $= 5$ and `n_units` $= 20$. This network has a competitive error for far less FLOPs and parameters.

Remark. *The obtained neural network is trained with* `local_variance = None`*. It is not contradictory with the previous results and is easily explained by the stochastic nature of deep learning, the sampling schemes and Bayesian optimization.*

6.4.3.2 Effect on the hybrid code

We now plug this neural network into the hybrid code to assess its prediction and to illustrate the benefits of HSIC and VBSW. To apply HSIC analysis and VBSW, we conduct the algorithm TS-GPBO described in Section 5, and we include variance-based sampling and weighting as hyperparameters. We compare the obtained network with the previous one. The

	MPP_high (ref)	MPP_low	NN_high	NN_low	NN_high	NN_low
method.	-	-	**thesis**	**thesis**	**vanilla**	**vanilla**
Time (s)	9478	4090	529	220	1209	531
Impr. (×)	-	2.3	17.9	43.1	7.8	17.9
Pressure						
L^2	-	3.19×10^{-5}	$\mathbf{1.13 \times 10^{-6}}$	4.00×10^{-5}	1.20×10^{-6}	3.23×10^{-5}
L^∞	-	9.07×10^{-3}	2.22×10^{-3}	9.82×10^{-3}	$\mathbf{2.00 \times 10^{-3}}$	9.35×10^{-3}
Shock dist.						
L^2	-	9.14×10^{-5}	$\mathbf{6.55 \times 10^{-6}}$	9.19×10^{-5}	8.54×10^{-6}	9.18×10^{-5}
L^∞	-	3.6×10^{-2}	1.08×10^{-2}	3.6×10^{-2}	$\mathbf{1.06 \times 10^{-2}}$	3.6×10^{-2}

Table 6.4: Execution times and normalized errors for the different codes after TS-GPBO, compared to random search (RS).

results are gathered in Table 6.4. We managed to obtain a ×2 speed-up for comparable and sometimes better prediction error, which illustrates the benefits of TS-GPBO.

6.4.4 Is the error acceptable?

We obtained satisfying results both with the **vanilla** and the **thesis** methodologies. However, the prediction of NN was assessed by computing its error with respect to the target obtained using MPP_high. The error of NN was comparable to that of MPP_low on this prediction, but it was on one single prediction. This evaluation process is not sufficient to state if the error is acceptable or not. In order to ensure prediction guarantees, which are mandatory for using codes in production, we have to go deeper into the analysis.

6.5 Guarantees for the hybrid code

In this section, we introduce two ways of obtaining prediction guarantees for NN. The first ensures to have exactly the same prediction accuracy as MPP but brings additional computations. The second compares the error with other sources of error to assess if it is acceptable, which allows using NN at full speed if it does.

6.5.1 Zero error guarantee using initialization

As we mentioned earlier, the solver used in MPP is iterative. It is initialized with a guess solution, which is usually uninformative - usually the same value over the entire mesh - and

iterations are made until a certain convergence criterion is reached. It is possible to first execute the hybrid code NN and to use its prediction as initialization for MPP. Then, MPP converges in fewer iterations, since the initialization is supposed to be close to the convergence point.

For the MPP_low code, once the prediction of NN_low is given as input to the original code, it reaches convergence in 163 seconds. If we add the execution time of the NN code, NN+MPP takes 383 seconds which is 10.6 times faster than MPP_low alone, for a prediction whose accuracy is ensured by the original code. In that case, there is no need to compare the pressure profile or the shock distance since **the prediction is exact**. In other words, we obtained an acceleration of a factor 10.6 for the exact same accuracy.

Remark. ***Generality of the approach:*** *This approach is not specific to our test case but can be applied to any simulations that involve iterative solvers and whose iterations are not of interest as simulation outputs. Such instances can be simulations of stationary phenomenon.*

6.5.2 Acceptable error guarantee using error analysis

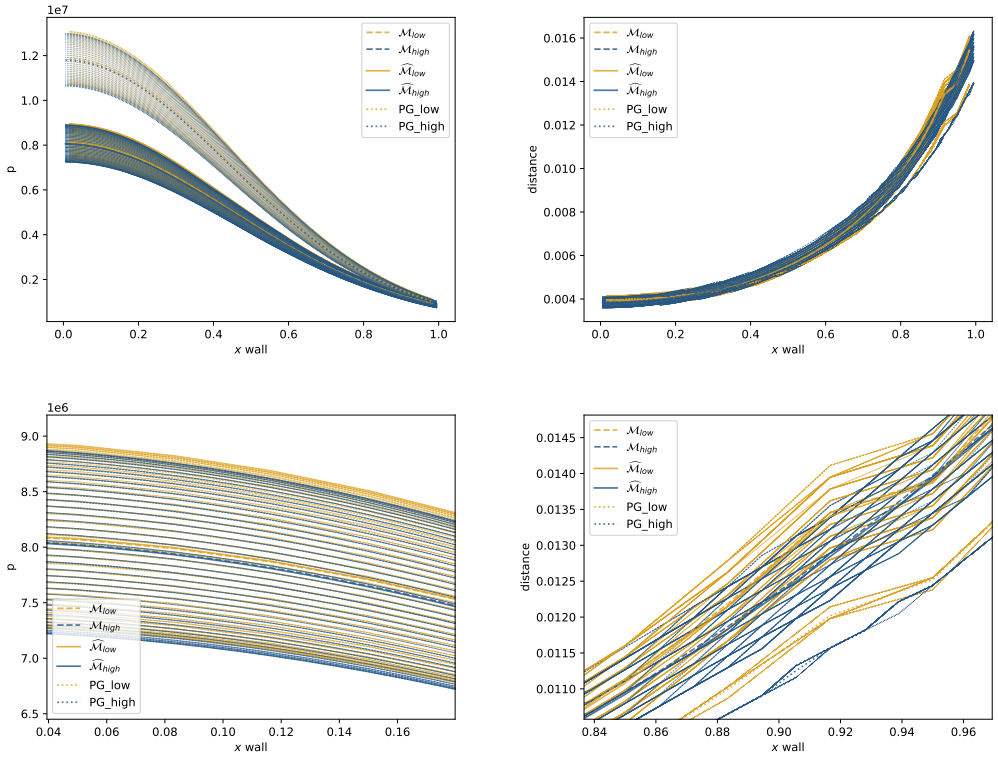
Using a neural network to design hybrid codes implies to add an error coming from the neural network approximation. However, there are several external, classical sources of error in the conception of the simulation codes. In this section, we compare the error of the hybrid code with these other errors.

We denote the original code MPP by $\mathcal{M}$ and the hybrid code NN by $\widehat{\mathcal{M}}$, with $\mathcal{M}_{high}$ denoting the model of MPP_high, and so on for the other MPP and NN codes. The models take parameters $\boldsymbol{x}$ as input and output a prediction $\mathcal{M}(\boldsymbol{x})$ and $\widehat{\mathcal{M}}(\boldsymbol{x})$. In our case, the input vector $\boldsymbol{x}$ contains the upstream pressure, temperature, and speed. The error of the predictions can be decomposed as

$$\begin{cases} e_{\mathcal{M}}(\boldsymbol{x}) = |\delta_{\Delta,\mathcal{M}}(\boldsymbol{x}) + \delta_{\boldsymbol{x},\mathcal{M}}(\boldsymbol{x})|, \\ e_{\widehat{\mathcal{M}}}(\boldsymbol{x}) = |\delta_{\Delta,\widehat{\mathcal{M}}}(\boldsymbol{x}) + \delta_{\boldsymbol{x},\widehat{\mathcal{M}}}(\boldsymbol{x}) + \delta_{\boldsymbol{\theta},\widehat{\mathcal{M}}}(\boldsymbol{x})|. \end{cases} \tag{6.6}$$

Equation (6.6) emphasizes three different types of errors :

- The discretization errors $\delta_{\Delta,\mathcal{M}}$ and $\delta_{\Delta,\widehat{\mathcal{M}}}$. The choice of the mesh used to run the simulation has an impact on the prediction error. A low mesh resolution may degrade the error, as we saw with MPP_low, but the geometry of the mesh also has its impact.
- The parameters' errors $\delta_{\boldsymbol{x},\mathcal{M}}$ and $\delta_{\boldsymbol{x},\widehat{\mathcal{M}}}$. In practice, we conduct numerical simulations because we are interested in the output of a phenomenon under specific conditions of interest. Nonetheless, we may have imperfect knowledge of these conditions of interest. This translates into uncertainties on the input vector $\boldsymbol{x}$, which contains the values of



(a) Pressure profile

(b) Shock distance

Figure 6.8: Pressure profile and shock distance projected on the wall of the object, for 40 different values of the upstream speed. In the bottom line, zoom of the curves in the highest error area.

the parameters that define the conditions of the simulation. These uncertainties have an impact on the model prediction, and therefore on their prediction error.

- The neural network approximation error $\delta_{\boldsymbol{\theta},\widehat{\mathcal{M}}}$. In the hybrid code, NN, the neural network approximates $\mathbf{M}_{++}$ with a certain error. This error propagates through the hybrid code, thereby affecting its prediction error.

In order to ensure that NN yields reliable predictions neural network approximation error must be compared with parameters' and discretization errors. If the former is at most similar to the latter, NN could be used safely.

Remark. *Another type of error is often specified when decomposing the error of numerical codes. This error is called modeling error and refers to the error that stems from modeling choices. In our case, the model error would be the error between PG and MPP, coming from the choice not to simulate the chemistry in PG. Another model error could come from the choice to simulate the chemistry with not more than three species.*

6.5.2.1 Parameters' error

To assess parameters' error, we introduce a perturbation in $\boldsymbol{x}$, modeling the uncertainty on the upstream speed. Let $\mathbf{x} = (\mathrm{x}, y, z)$, with $\mathrm{x} \sim \mathcal{U}(0.95x, 1.05x)$, $x = 4930.83$ (the nominal value of the initial test case). The random variable x traduces the uncertainty on the speed of the upstream field, and the values of y and z are the upstream pressure and temperature. We simulate the random variable x on $N = 40$ Gauss quadrature points $\{x_1, ..., x_N\}$, with $x_i \in [0.95x, 1.05x]$, that are used to evaluate $\mathbb{E}[\mathcal{M}(\boldsymbol{x})]$ with $\mathcal{M} \in \{\mathcal{M}_{low}, \mathcal{M}_{high}, \widehat{\mathcal{M}}_{low}, \widehat{\mathcal{M}}_{high}\}$. The mean $\mathbb{E}[\mathcal{M}(\boldsymbol{x})]$ and each of the 40 curves are plotted in Figure 6.8. These plots are quite loaded, but they emphasize that the variability induced by parameters uncertainty is much higher than that coming from approximation, and even discretization errors.

Benefits of NN+MPP

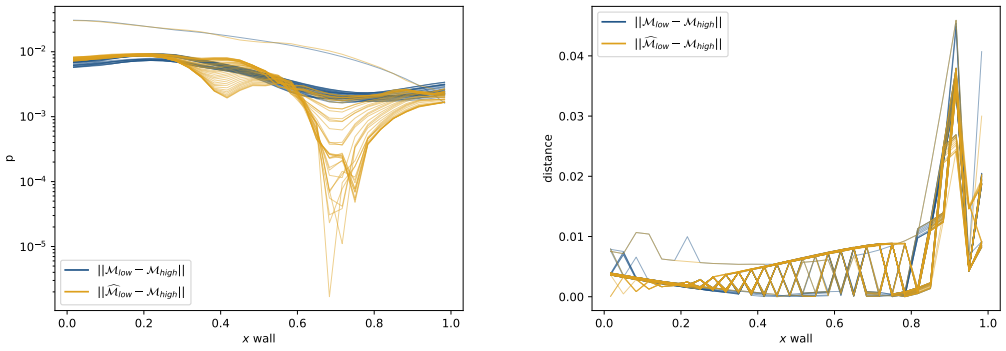
To conduct this experiment, we never executed $\mathcal{M}_{low}$ and $\mathcal{M}_{high}$ entirely, but always used an initialization from the prediction of $\widehat{\mathcal{M}}_{low}$ and $\widehat{\mathcal{M}}_{high}$ (NN+MPP, as described in Section 6.5.1). The advantages were twofold. First, the study was much faster (approximately by a factor of 10). Second, for some $\{x_1, ..., x_N\}$, MPP did not converge, perhaps because of numerical instabilities. Initializing MPP using the hybrid code solved this problem. This echoes the remark of Section 6.4.2.1 on the artefacts in predictions of C_p and C_v by $\mathbf{M}_{++}$.

6.5.2.2 Discretization error

Comparing discretization and neural network approximation errors is more subtle because it is not clear-cut in Figure 6.8. To do so, we directly compare the discretization error of $\mathcal{M}_{low}$ with the approximation error of $\widehat{\mathcal{M}}_{low}$ and $\widehat{\mathcal{M}}_{high}$ under parameters uncertainty. First, we plot $||\mathcal{M}_{low} - \mathcal{M}_{high}||$ and $||\widehat{\mathcal{M}}_{low} - \mathcal{M}_{high}||$ for the 40 values of x_i in Figure 6.9. Here, $||.||$ is the normalized absolute difference evaluated point-wise in the pressure profile and the shock distance. At first sight, the approximation error seems to be lower, for the pressure profile, and equivalent, for the shock distance, to the discretization error.

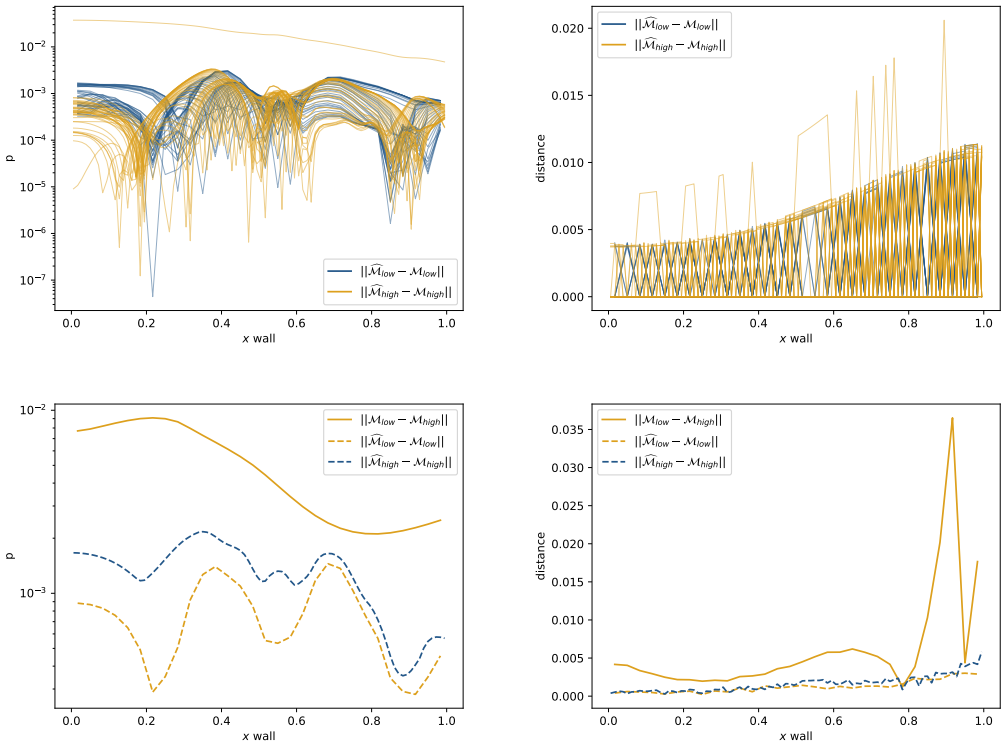
We confirm this observation by plotting $||\widehat{\mathcal{M}}_{low} - \mathcal{M}_{low}||$ and $||\widehat{\mathcal{M}}_{high} - \mathcal{M}_{high}||$ in Figure 6.10. We plot the 40 curves corresponding to each x_i and the mean estimated using the Gauss quadrature. It clarifies the comparison and strengthens the conclusion that approximation error is lower than both discretization and parameters error. Note that in the pressure profile, one x_i value leads to outlying errors for $\widehat{\mathcal{M}}_{high}$, $\widehat{\mathcal{M}}_{low}$ and $\mathcal{M}_{low}$. This value of i is 13, the famous lucky number, which is worth mentioning. More seriously, we did not explain the origin of this outlying value, but we can at least say that it does not come from the neural network approximation error since even the prediction of $\mathcal{M}_{low}$ is abnormally erroneous.

This error study shows that neural network approximation error can be small when compared to other types of errors. In this case, the hybrid code is reliable, which is a strong argument



(a) Pressure profile (b) Shock distance

Figure 6.9: Discretization errors $||\mathcal{M}_{low} - \mathcal{M}_{high}||$ and $||\widehat{\mathcal{M}}_{low} - \mathcal{M}_{high}||$ for the pressure profile and the shock distance for each of the 40 different values of the upstream speed.



(a) Pressure profile (b) Shock distance

Figure 6.10: Top: approximation errors $||\widehat{\mathcal{M}}_{low} - \mathcal{M}_{low}||$ and $||\widehat{\mathcal{M}}_{high} - \mathcal{M}_{high}||$ for the pressure profile and the shock distance for each of the 40 different values of the upstream speed. Bottom: comparison of the mean of the discretization and approximation errors

in favor of the use of hybrid simulation codes.

6.6 Discussion and Perspectives

The perspectives that follow this work are numerous and emphasize the potential of the presented methodology. These perspectives are twofold. First, the specific test case of approximating Mutation++ could be explored more thoroughly since this library is used in many simulation codes. Second, the methodology applied is not specific to Mutation++, and its success is an argument in favor of more general usage of hybrid simulation codes for numerical simulations.

6.6.1 Towards a general approximation of Mutation++

In this chapter, the methodology for approximating Mutation++ consists of constructing a training database and fitting a neural network. However, the neural network is trained for a fixed output dimension corresponding to the number of species. It cannot be used for test cases that involve different species. This constraint is an obstacle towards a general usage of neural networks to approximate Mutation++. Indeed, it requires constructing a new training database for each different chemical setting.

In order to overcome this problem, it would be interesting to investigate the use of transfer learning. The idea behind transfer learning is to use a neural network trained for one task in another similar task. When there is a large training database for the first task but only limited data for the other task, it allows obtaining good results on the second task despite the data frugality. It is intensively used in computer vision, and language understanding, where neural networks pre-trained on Imagenet (with more than one million images and a thousand classes) or on Wikipedia text database are fine-tuned on more specific tasks. In our case, we could pre-train a neural network once for a high number of different species and with a large database constructed out of Mutation++. Then, we could find a simple way to adapt this neural network for each different test case, for instance, with a simple least-squares linear regression on the feature space of the pre-trained network (like we did in 3.5), using a smaller data set.

6.6.2 Possible usages of hybrid simulation codes

This chapter introduced a methodology for constructing hybrid numerical simulation codes, which brings significant improvements in terms of speed-up. Moreover, the error induced by the neural network approximation at play when constructing the hybrid code can be negligible compared to other classical errors of numerical simulations. These results unlock many perspectives.

Prediction speed-up. When the neural network approximation error is negligible, which can be checked with an error analysis similar to Section 6.5, the hybrid code can be used at its quickest form. In that case, it provides reliable predictions with a similar accuracy than the original code, for at least a decade lower execution time.

When the error analysis concludes that the approximation error is not negligible and not acceptable, it is always possible to initialize the numerical simulation code with the prediction of the hybrid code (NN+MPP). This process yields a prediction whose accuracy is ensured by the original simulation code. To sum up, constructing hybrid codes almost always grants a speed-up for a single prediction, which translates into comparable time savings when using these codes in production.

The benefits of these time savings can go even further. Indeed, some test cases are so computationally intensive that they simply cannot be executed. Using hybrid codes could help to close the gap of this infeasibility: finer physics might no longer be out of reach.

Parametric studies. The use for which the prediction speed-ups are the most promising is parametric studies, like uncertainty quantification, sensitivity analysis and calibration (like in we demonstrated in Section 6.5). Indeed, using hybrid codes in place of classical surrogate models could lead to far more accurate estimation since surrogate models are usually trained on a limited set of data.

Such parametric studies may require an unaffordable number of predictions, even when using a hybrid code, making the use of surrogate models mandatory. Nonetheless, in these cases, the hybrid code could be used to generate a more furnished training database, thereby improving the performances of the surrogate model and the quality of the parametric study.

Finally, numerical simulation codes may be so expensive that even the construction of a minimal training data set for surrogate modeling would be out of reach, forbidding any parametric study. Using hybrid codes could enable parametric studies for such test cases.

Decision-making. There is always interest in speeding up a numerical simulation code used for production or parametric studies. However, a given code is often not a final product, and research and engineering efforts are continuously produced to improve it. The construction of hybrid codes may help to make decisions about the orientation of these efforts.

Constructing a hybrid code with our methodology implies approximating a part of the simulation with a neural network. There may be questions about how to include this part in the simulation. For instance, in our case, we could wonder if it is worth simulating chemical equilibrium in the simulation and, if so, to what precision. Comparing the code PG (without chemical equilibrium) and NN_low shows that simulating chemical equilibrium leads to significantly different predictions, which answers this question. Still, we could go further and study how many species this simulation requires to produce accurate predictions. These conclusions could drive efficient researches towards more and more accurate simulation codes.

Finally, parametric studies are also intensively used for decision-making. Indeed, the results of these studies always lead to design choices, which directly impact production.

6.6.3 Concluding remark

We would like to point out that even if the obtained hybrid code no longer uses the code part that is approximated by a neural network, this part is still crucial for constructing the hybrid code. Indeed, a good training set is mandatory to ensure neural network accuracy, and the original code part is key to achieve that. That is why we do not claim supervised deep learning to replace certain simulation codes. Rather, it should be seen as an additional step in constructing simulation codes that allows for significant accelerations.

Chapter 7

Summary of the thesis/Résumé de la thèse (in French)

7.1 Introduction

Les innovations récentes en mathématiques, en informatique et en ingénierie ont permis de réaliser des simulations numériques de plus en plus complexes. Cependant, certaines simulations restent inabordables en termes de temps de calcul, même pour les supercalculateurs les plus puissants. Récemment, l'apprentissage automatique a démontré sa capacité à améliorer l'état de l'art dans de nombreux domaines, notamment la vision par ordinateur, la compréhension du langage et la robotique. Cette thèse s'inscrit dans le domaine émergent et à fort enjeu de l'apprentissage automatique scientifique, qui étudie l'application de l'apprentissage automatique au calcul scientifique. Plus précisément, nous nous intéressons à l'utilisation de l'apprentissage profond pour accélérer les simulations numériques. Ce faisant, nous mettons en évidence certains liens entre ces deux domaines, en présentant des contributions ayant un impact sur l'un comme sur l'autre.

Contents

7.1.1 L'apprentissage automatique comme réponse aux enjeux de la simulation numérique

Le domaine de la simulation numérique se retrouve toujours soumis à un compromis entre précision et performances. En effet, le but des simulations est de reproduire des phénomènes physiques le plus fidèlement possible à l'aide de programmes informatiques. Cependant, améliorer la fidélité des simulations implique de réaliser des calculs beaucoup plus couteux. Il est donc nécessaire de trouver un compromis pour que le programme s'exécute en un temps raisonnable sans trop simplifier le phénomène simulé.

Dans cette thèse, nous étudions l'utilisation de l'apprentissage profond pour accélérer des codes de simulation. Bien souvent, ces codes de simulation sont constitués de couplages entre différents systèmes d'Équations aux Dérivées Partielles (EDP) simulant chacun un phénomène différent. L'approche retenue consiste à l'approximation d'un de ces systèmes d'EDP avec un réseau de neurones, puis au couplage du réseau de neurones obtenu avec le reste du code.

Cette méthode présente plusieurs avantages. Premièrement, elle permet d'éviter d'avoir à réaliser des hypothèses simplificatrices sur la physique du problème. Ensuite, ne procéder à l'approximation que d'une partie du code, et non du code en entier, permet de constituer une base d'apprentissage plus fournie, ce qui favorise la précision du réseau de neurones. Enfin, l'implémentation du réseau de neurones présente beaucoup d'avantages : son exécution revient à une succession d'opérations vectorielles, ce qui permet de traiter naturellement beaucoup de données en parallèle; la complexité de son exécution est linéaire avec les dimensions du problème et ne dépend pas du nombre de points utilisés pour l'entraînement.

7.1.2 Interactions entre simulation numérique et apprentissage profond supervisé

Dans notre contexte, utiliser la méthode de l'apprentissage profond pour l'approximation de fonction se fait en trois étapes:

1 construire une base de données d'entraînement,

2 définir l'architecture du réseau de neurones,

3 procéder à son entraînement.

Chacune de ces étapes revêt une importance particulière si elles sont considérées à travers le prisme du compromis précision-performances. En effet, une base de données d'entraînement bien construite permet d'améliorer la précision d'un réseau de neurones sans sacrifier les performances. Ensuite, l'architecture d'un réseau de neurones doit être choisie avec soin car elle a un fort impact à la fois sur sa précision et ses performances. Enfin, la qualité de l'entraînement du réseau de neurones conditionne fortement la précision de l'approximation, sans affecter les performances. Cette thèse propose donc d'étudier chacune de ces étapes.

Les enjeux de ce travail pour la simulation numérique se résument à l'amélioration du compromis précision-performances. Cependant, aborder la méthodologie de l'apprentissage profond dans cette optique permet également de tirer des bénéfices pour la pratique de l'apprentissage profond en général. Les contributions de cette thèse profitent donc à la fois au domaine de la simulation numérique et au domaine de l'apprentissage profond.

7.1.3 Organisation de la thèse

Dans un premier temps, nous rappelons les bases de l'apprentissage profond supervisé, nécessaires à la compréhension des travaux réalisés dans cette thèse. Ensuite, nous abordons successivement chacune des trois étapes présentées au paragraphe précédent dans des sections dédiées. Enfin, nous consacrons une dernière section à l'utilisation de ces contributions pour la construction d'un code hybride dans le cadre d'une simulation d'un écoulement autours d'un objet prenant en compte des réactions chimiques dans le fluide.

7.2 Bases de l'apprentissage profond supervisé

Cette partie décrit la façon dont sont construits les réseaux de neurones. Il comprend également des rappels sur l'apprentissage profond supervisé.

7.2.1 Les réseaux de neurones

Un réseau de neurones simple (MLP, pour Multi Layer Perceptron en anglais) consiste en une succession de couches de neurones. Il peut être complètement décrit par deux ensembles de paramètres. Le premier décrit sa forme, appelée architecture :

- d la profondeur du réseau,
- n_k le nombre de neurones sur la k-ième couche, avec $k \in \{1, d+2\}$ (avec $n_{d+1} = n_{out}$ et $n_0 = n_{in}$),
- σ_k la fonction d'activation de la k-ième couche, généralement non linéaire.

Ces paramètres font partie d'un ensemble dont les membres sont appelés hyperparamètres.Cet ensemble définit l'architecture des réseaux de neurones ainsi que les conditions de leur entraînement. Le deuxième ensemble de paramètres, correspondant à $\boldsymbol{\theta}$, contient les paramètres à optimiser pendant le processus d'apprentissage :

- $\boldsymbol{W}^k = \{\omega_{ij}\}, (i,j) \in \{0, n_{k+1}-1\} \times \{0, n_k - 1\}$ la matrice des poids entre les $(k-1)$-ième et k-ième couches,
- $\mathbf{b}^k = (b_0, ..., b_{n_k - 1})$ le vecteur de bais de la k-ième couche.

Soient $f^k : \mathbb{R}^{n_k} \to \mathbb{R}^{n_{k+1}}$ telle que

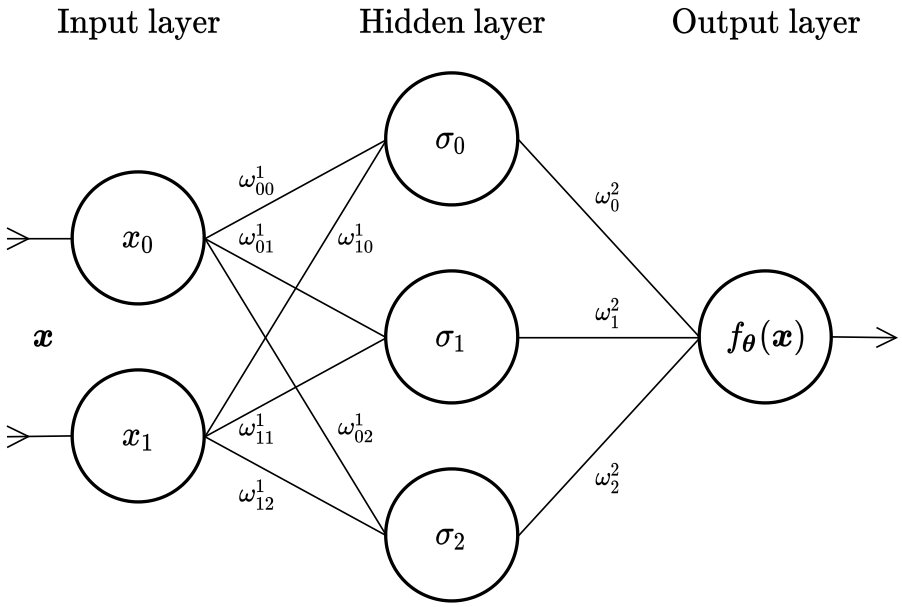


Figure 7.1: Réseau de neurones de profondeur $d = 1$, avec $n_{in} = n_1 = 2$, $n_2 = 3$ et $n_{out} = n_3 = 1$. Dans ce cas précis, pour des raisons de lisibilité, le i-ième composant de la sortie de la première couche est notée σ_i.

$$f_i^k(\boldsymbol{x}) = \sigma_k\Big(\sum_{j=0}^{j=n_k-1} \omega_{ij}^k x_j + b_i^k\Big) = \sigma_k(\boldsymbol{W}^k\boldsymbol{x} + \boldsymbol{x}^k) \tag{7.1}$$

avec $\boldsymbol{x} = (x_1, ..., x_{n_k}) \in \mathbb{R}^{n_k}$. Alors, $f_{\boldsymbol{\theta}} : \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}}$, la fonction du réseau de neurones peut s'écrire

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = f^d \circ \cdots \circ f^1(\boldsymbol{x}) \tag{7.2}$$

avec $\boldsymbol{\theta} = \{\boldsymbol{W}^k, b_k | k \in \{1, d+1\}\}$. L'equation (7.1) et l'equation (7.2) mettent en évidence qu'un réseau de neurones peut être vu comme une succession de produits matrices-vecteurs. Sur la figure 7.1 est repréenté un réseau de neurones de profondeur $d = 1$, avec $n_{in} = 2$, $n_k = 3$ et $n_{out} = 1$. Pour ce réseau de neurones, nous avons

$$\begin{cases} f_{\boldsymbol{\theta}}(\boldsymbol{x}) & = \omega_0^2\sigma(\omega_{00}^1 x_0 + \omega_{10}^1 x_1 + b_0^1) + \omega_1^2\sigma(\omega_{01}^1 x_0 + \omega_{11}^1 x_1 + b_1^1) + \omega_2^2\sigma(\omega_{02}^1 x_0 + \omega_{12}^1 x_1 + b_2^1) + b^2 \\ & = \sum_{i=0}^{2} \omega_i^2\sigma(\sum_{j=0}^{1} \omega_{ji}^1 x_j + b_i) + b^2 \\ \boldsymbol{\theta} & = \{\omega_0^2, \omega_{00}^1, \omega_{10}^1, \omega_1^2, \omega_{01}^1, \omega_{11}^1, \omega_2^2, \omega_{02}^1, \omega_{12}^1, b_0^1, b_1^1, b_2^1, b^2\} \end{cases}$$

7.2.2 L'apprentissage profond supervisé

L'apprentissage supervisé consiste à apprendre une fonction reliant des données d'entrée à des données de sortie. Pour cela, un modèle paramétrique est construit et optimisé pour reproduire cette fonction le mieux possible.

Plus précisément, nous notons cette fonction $f : \mathbf{S} \subset \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}}$ où $\mathbf{S}$ est un sous espace mesurable de $\mathbb{R}^{n_{in}}$ qui dépend de l'application considérée. L'apprentissage revient donc à effectuer une approximation de f à l'aide d'un modèle d'apprentissage, noté $f_{\boldsymbol{\theta}}$, où $\boldsymbol{\theta} \in \Theta$ contient tous les paramètres du modèle. Optimiser $f_{\boldsymbol{\theta}}$ pour effectuer une approximation de f signifie trouver une valeur $\boldsymbol{\theta}^*$ pour $\boldsymbol{\theta}$ qui minimise une fonction de coût intégrée $J_{\mathbf{x}}(\boldsymbol{\theta}) = \mathbb{E}[L(f_{\boldsymbol{\theta}}(\boldsymbol{x}), f(\boldsymbol{x}))]$, où L est une fonction de coût, $L : \mathbb{R}^{n_o} \times \mathbb{R}^{n_o} \to \mathbb{R}$. La variable aléatoire $\mathbf{x} \sim \mathbb{P}_{\mathbf{x}}$ traduit la distribution de probabilité des données d'entrée. Pour un réseau de neurones tel que présenté dans le paragraphe précédent, $\boldsymbol{\theta} = \{\boldsymbol{W}^k, b_k | k \in \{0, d\}\}$.

En pratique, nous n'avons pas accès à $\mathbb{P}_{\mathbf{x}}$ et pour optimiser $f_{\boldsymbol{\theta}}$, $J_{\mathbf{x}}$ est estimé en utilisant N points $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\} \in \mathbf{S}$ qui sont des échantillons de $\mathbf{x}$, et leur image par f $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$.

Le modèle $f_{\boldsymbol{\theta}}$ est ensuite entraîné avec un algorithme d'optimisation. Dans le cas des réseaux de neurones, l'algorithme le plus utilisé est la descente de gradient stochastique (SGD, Robbins and Monro (1951)), dont l'implémentation est rendue très efficace par l'algorithme de rétro-propagation du gradient, basée sur la différentiation automatique.

7.3 Construction d'une base de données d'entraînement à partir des variations locales

Dans cette partie, nous nous intéressons à la construction de la base de données d'entraînement pour l'apprentissage profond supervisé. En calcul scientifique, le cadre utilisé pour construire une telle base de données est appelé plan d'expériences. Plusieurs méthodes existent déjà, comme Latin Hypercube Sampling (McKay et al., 1979), Maximin margins (Johnson et al., 1990), ou encore plan d'entropie maximum (Shewry and Wynn, 1987), mais elles reposent principalement sur l'idée de répartir les points d'apprentissage de la manière la plus homogène possible sur le domaine d'entraînement.

Ici, nous abordons le problème en suivant une autre approche. Nous partons de l'observation qu'un réseau de neurones aura plus de difficulté dans l'approximation d'une fonction f aux endroits où cette fonction connaît de fortes variations. Celui-ci aura donc besoin de plus de points d'apprentissage dans ces régions de l'espace.

7.3.1 Échantillonnage à partir du développement de Taylor

Nous construisons une nouvelle distribution d'entraînement, $d\mathbb{P}_{\bar{\mathbf{x}}}$, qui focalise l'échantillonnage de la base de données d'entraînement sur les zones où f varie fortement. Il reste à déterminer comment identifier ces zones. Pour cela, nous utilisons le développement de Taylor de la fonction f. Ce développement est valable au voisinage d'un point $\boldsymbol{x} \in \mathbf{S}$, où $\mathbf{S}$ est le domaine de définition de f, et implique les dérivées de f. Il relie donc les variations de f (via les dérivées) à une zone précise (le voisinage de $\boldsymbol{x}$). A partir de ce développement, nous construisons un indice qui est proportionnel à l'amplitude des variations locales de f dans le voisinage de $\boldsymbol{x}$. Cet indice s'écrit

$$Df_{\boldsymbol{\epsilon}}^{n}(\boldsymbol{x}) = \sum_{1 \leq |\boldsymbol{k}| \leq n} \frac{\|\boldsymbol{\epsilon}\|^{k} \cdot \| \operatorname{Vect}(\partial^{\boldsymbol{k}} f(\boldsymbol{x}))\|}{\boldsymbol{k}!}, \tag{7.3}$$

où $\partial^{\boldsymbol{k}} f$ correspond à la notation multi-indices des dérivées d'ordre k de f et $\boldsymbol{\epsilon}$ indique la taille du voisinage de $\boldsymbol{x}$ considéré. En pratique, nous n'utiliserons que le développement à l'ordre $n = 2$, c'est-à-dire $Df_{\boldsymbol{\epsilon}}^{2}(\boldsymbol{x})$. Il est donc possible de construire une méthodologie d'échantillonnage que nous appelons échantillonnage par développement de Taylor (noté TBS). Cette méthodologie est récapitulée dans l'Algorithme 6.

Line 1-2: La méthodologie nécessite une initialisation basée sur l'échantillonnage uniforme de N points. Ensuite, N' nouveaux points seront ajoutés à la base d'entraînement. **Line 3-5:** La construction de $\{Df_{\boldsymbol{\epsilon}}^{n}(\boldsymbol{x}_1), ..., Df_{\boldsymbol{\epsilon}}^{n}(\boldsymbol{x}_N)\}$ se fait à partir de ces points initiaux. **Line 6:** L'échantillonnage à partir de $Df_{\boldsymbol{\epsilon}}^{n}$ peut se faire grâce à des méthodes d'approximation de densité. Dans ce cas, nous avons utilisé un mélange de Gaussiennes. **Line 7-8:** Il reste à évaluer l'image de la fonction en les nouveaux points obtenus après l'échantillonnage, pour compléter la base de données d'entraînement finale comportant $N + N'$ points.

Algorithm 6 Taylor Based Sampling (TBS)

1: **Entrées:** $\boldsymbol{\epsilon}$, N, N'
2: Échantillonner $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ à partir de $\mathbf{x} \sim \mathcal{U}(\mathbf{S})$
3: **for** $0 \leq k \leq n$ **do**
4: Calculer $\{\partial^{\boldsymbol{k}} f(\boldsymbol{x}_1), ..., \partial^{\boldsymbol{k}} f(\boldsymbol{x}_N)\}$
5: Calculer $\{Df_{\boldsymbol{\epsilon}}^{n}(\boldsymbol{x}_1), ..., Df_{\boldsymbol{\epsilon}}^{n}(\boldsymbol{x}_N)\}$ en utilisant equation (7.3)
6: Échantillonner $\{\bar{\boldsymbol{x}}_1, ..., \bar{\boldsymbol{x}}_{N'}\}$ à partir de $d\mathbb{P}_{\bar{\mathbf{x}}} \propto Df_{\boldsymbol{\epsilon}}^{n}$
7: Calculer $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$
8: Ajouter $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$ à $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$

Sur la figure 7.2, nous illustrons $\boldsymbol{x} \to Df_{\boldsymbol{\epsilon}}^{2}(\boldsymbol{x})$ avec f définie par la fonction de Runge, puis par la fonction tangente hyperbolique. Nous faisons également figurer les points supplémentaires échantillonnés à partir de TBS.

Le fait de modifier la distribution d'entraînement ressemble à certaines méthodes employées

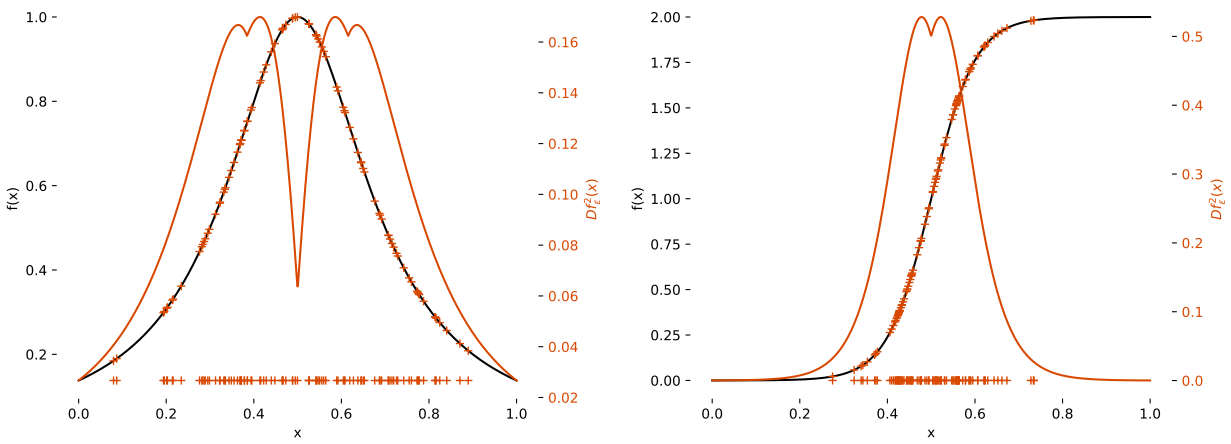


Figure 7.2: **Gauche :** (Axe de gauche) fonction de Runge par rapport à x et (axe de droite) $x \to Df_{\boldsymbol{\epsilon}}^2(x)$. Les points échantillonnés selon TBS sont placés sur l'axe des abscisses et projetés sur f. **Droite :** Même figure avec la fonction f tangente hyperbolique.

en apprentissage profond, comme l'apprentissage par cursus (Bengio et al., 2009b) ou encore la recherche d'exemples complexes (Shrivastava et al., 2016). TBS se démarque par une caractérisation mathématique de l'échantillonnage, moins heuristique que ces dernières méthodes.

7.3.2 Généralisation par pondération basée sur la variance locale

Tel quel, TBS ne peut pas être appliqué pour des problèmes génériques d'apprentissage profond. En effet, en général, la fonction f n'est pas connue. Cela implique qu'il est impossible d'évaluer ses dérivées, et donc $Df_{\boldsymbol{\epsilon}}^2(x)$. De plus, même en supposant que l'on puisse échantillonner de nouveaux points d'apprentissage $\{\bar{\boldsymbol{x}}_1, ..., \bar{\boldsymbol{x}}_{N'}\}$, il serait impossible de calculer $\{f(\bar{\boldsymbol{x}}_1), ..., f(\bar{\boldsymbol{x}}_{N'})\}$.

Pour régler le problème des dérivées, nous définissons de nouveaux indices :

$$\widehat{Df^2}(\boldsymbol{x}_i) = \frac{1}{k-1} \sum_{\boldsymbol{x}_j \in \mathcal{S}_k(\boldsymbol{x}_i)} \left(f(\boldsymbol{x}_j) - \frac{1}{k} \sum_{\boldsymbol{x}_l \in \mathcal{S}_k(\boldsymbol{x}_i)}^{k} f(\boldsymbol{x}_l) \right)^2, \tag{7.4}$$

où $\mathcal{S}_k(\boldsymbol{x}_i)$ est l'ensemble des k plus proches voisins de $\boldsymbol{x}$. Cet indice repose sur une estimation de la variance de f dans le voisinage de $\boldsymbol{x}$. Il est démontré dans la thèse que lorsque $||\boldsymbol{\epsilon}|| \to 0$, la variance locale est un estimateur de $Df_{\boldsymbol{\epsilon}}^2(\boldsymbol{x})$.

Enfin, pour palier le problème de l'échantillonnage, nous remplaçons ce dernier par une pondération. Il s'agit de pondérer chaque point d'entraînement $\boldsymbol{x}_i$ par $\widehat{Df_{\boldsymbol{\epsilon}}^2}(\boldsymbol{x}_i)$. Cet algorithme est appelé pondération des points par la variance locale (VBSW, pour Variance based Sample

Weighting en anglais).

7.3.3 Résultats

Les améliorations découlant de TBS et VBSW sont démontrées sur différents cas d'application. TBS est étudié sur l'approximation de la résolution d'un système d'équations de Bateman, décrit en annexe (cf **Appendix C**) et VBSW est étudié sur des problèmes de classification d'image et de classification/régression de texte.

7.3.3.1 Équations de Bateman

Pour ce cas test, un simple MLP est utilisé. L'expérience est répétée 50 fois et l'erreur L_2 moyenne est affichée avec son erreur standard d'estimation. L'erreur L_∞ est aussi représentée. Les résultats sont rassemblés dans le tableau 7.1.

Sampling	L_2 error ($\times 10^{-4}$)	L_∞ ($\times 10^{-1}$)
Uniforme	1.22 ± 0.13	5.28 ± 0.47
TBS	$\mathbf{1.14} \pm 0.15$	$\mathbf{2.96} \pm 0.37$

Table 7.1: Comparaison entre l'échantillonnage uniforme et TBS pour les équations de Bateman

7.3.3.2 Classification d'image

Pour ce cas test, les cas d'application retenus sont MNIST et Cifar10, sur lesquels des réseaux de neurone de type Lenet (Lecun et al., 1998) et Resnet (He et al., 2015) sont respectivement entraînés. La grandeur d'évaluation considérée est la précision de classification. Celle-ci est présentée sous la forme : "précision max, précision moyenne ± erreur standard d'estimation de la moyenne". Les résultats sont rassemblés dans le tableau 7.2.

	VBSW	simple entraînement
MNIST	**99.09**, $\mathbf{98.87} \pm 0.01$	98.99, 98.84 ± 0.01
Cifar10	**91.30**, $\mathbf{90.64} \pm 0.07$	91.01, 90.46 ± 0.10

Table 7.2: Comparaison entre un simple entraînement et un entraînement avec VBSW pour MNIST et Cifar10

	VBSW		simple entraînement	
	m1	m2	m1	m2
RTE	**61.73**, **58.46** $\pm$ 0.15	-	61.01, 58.09 $\pm$ 0.13	-
STS-B	**62.31**, **62.20** $\pm$ 0.01	**60.99**, 60.88 $\pm$ 0.01	61.88, 61.87 $\pm$ 0.01	60.98, **60.92** $\pm$ 0.01
MRPC	**72.30**, **71.71** $\pm$ 0.03	**82.64**, **80.72** $\pm$ 0.05	71.56, 70.92 $\pm$ 0.03	81.41, 80.02 $\pm$ 0.07

Table 7.3: Comparaison entre un simple entraînement et un entraînement avec VBSW pour RTE, STS-B et MRPC. Pour RTE, le critère utilisé (m1) est la précision. Pour MRPC, deux critères sont utilisés : (m1) est la précision et (m2) est le score F1. Pour STS-B, (m1) est la corrélation de Spearman et (m2) est la corrélation de Pearson.

7.3.3.3 Classification et régression de texte

Pour ce cas test, les cas d'application retenus sont RTE, STS-B et MRPC, des jeux de données issus du benchmark Glue (Wang et al., 2019). Un réseau de neurones spécialisé dans l'analyse de données textuelles, Bert (Devlin et al., 2019), est utilisé. Le critère d'évaluation change pour chaque cas et est précisé dans la légende du tableau 7.3. Les résultats sont rassemblés dans ce tableau et sont présentés de la même manière que pour la classification d'image.

Ces résultats valident la méthode d'échantillonnage (TBS) et de pondération (VBSW), ainsi que l'observation initiale sur la difficulté d'apprentissage des réseaux de neurones lorsque f connaît de fortes variations. Ces conclusions sont intéressantes à la fois pour la simulation numérique, car il s'agit d'une nouvelle façon d'aborder les plans d'expérience, mais aussi pour l'apprentissage automatique en général, puisque VBSW peut s'appliquer facilement à un large panel de problèmes. De plus, cette vision de la difficulté d'apprentissage pourrait motiver d'autres futurs travaux.

7.4 Optimisation des hyperparamètres à l'aide de l'analyse de sensibilité globale

Dans la partie précédente, nous avons étudié la construction de la base de données d'entraînement, qui est la première étape de la méthodologie d'apprentissage profond supervisé. L'étape suivante est le choix de l'architecture du réseau de neurones, par le choix de ses hyperparamètres. Cette étape est cruciale, car les hyperparamètres d'un réseau de neurones ont un impact significatif sur la précision du réseau de neurones ainsi que sur son coût d'exécution. Or, les méthodes habituelles d'optimisation d'hyperparamètres (recherche aléatoire (Bergstra and Bengio, 2012), optimisation Bayesienne (Mockus, 1974), recherche d'architecture neuronale (Elsken et al., 2019), Hyperband (Li et al., 2018b) ...) ne sont pas naturellement conçues pour intégrer le coût d'exécution comme critère d'optimisation. De plus, ces méthodes suivent des approches boîte-noire, et l'utilisateur ne retire aucune interprétation ou justification du

résultat, ni de connaissances sur le comportement de l'erreur vis-à-vis des hyperparamètres.

Pour palier ce problème, nous introduisons l'utilisation des récentes avancées en analyse de sensibilité dans le cadre de l'optimisation des hyperparamètres. Plus précisément, nous nous focalisons sur l'analyse de sensibilité globale et orientée-objectif. Pour cela, nous utilisons un indice de sensibilité, le critère d'indépendance de Hilbert-Schmidt (HSIC, pour Hilbert-Schmidt Independance Criterion), et son dérivé adapté à l'analyse de sensibilité orientée-objectif.

7.4.1 Adaptation du critère d'indépendance de Hilbert-Schmidt

Les hyperparamètres des réseaux de neurones se caractérisent par leur diversité et la complexité de leurs interactions. Un exemple d'hyperparamètres définissant un réseau de neurones et son entraînement est donné en figure 7.3. Il est donc nécessaire d'adapter les outils d'analyse de sensibilité existants avant de les appliquer au problème d'optimisation d'hyperparamètres.

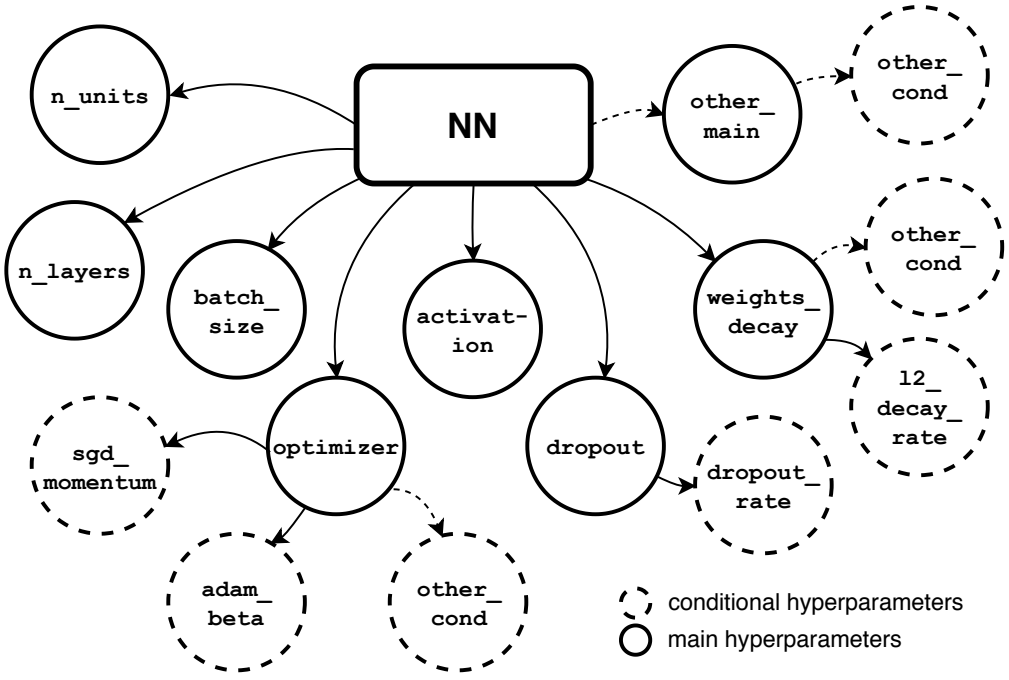


Figure 7.3: Exemple d'hyperparamètres définissant un réseau de neurones et son entraînement.

7.4.1.1 Définition du HSIC pour l'analyse de sensibilité orientée-objectif

Soient $\mathbf{x}$ et $\mathbf{y}$ deux variables aléatoires ayant pour distributions de probabilité $d\mathbb{P}_{\mathbf{x}}$ et $d\mathbb{P}_{\mathbf{y}}$. Le HSIC est construit à partir d'une distance mettant en jeu les projections de $d\mathbb{P}_{\mathbf{xy}}$ et $d\mathbb{P}_{\mathbf{x}}d\mathbb{P}_{\mathbf{y}}$ dans un espace de Hilbert restreint à noyaux reproductibles (RKHS, pour Reproducing Kernel Hilbert Space en anglais) de noyau k. Cette distance est appelée divergence maximale de la moyenne (MMD, pour Maximum Mean Discrepancy). Le HSIC s'écrit

$$HSIC(\mathbf{x}, \mathbf{y}) = \gamma_k^2(\mathbb{P}_{\mathbf{xy}}, \mathbb{P}_{\mathbf{y}}\mathbb{P}_{\mathbf{x}}), \tag{7.5}$$

où γ_k est la MMD. Cet indice est un critère d'indépendance car lorsque $\mathbf{x}$ et $\mathbf{y}$ sont indépendantes, $d\mathbb{P}_{\mathbf{xy}} = d\mathbb{P}_{\mathbf{x}} d\mathbb{P}_{\mathbf{y}}$, et donc $HSIC(\mathbf{x}, \mathbf{y}) = 0$. Dans le cas où $\mathbf{x}$ est un hyperparamètre et $\mathbf{y}$ l'erreur du réseau de neurones. Alors, si $HSIC(\mathbf{x}, \mathbf{y})$ est élevé, $\mathbf{x}$ a un impact sur l'erreur et si $\mathbf{y}$ est faible, cet impact est limité. Cependant, dans le cadre de l'optimisation des hyperparamètres, nous ne sommes pas intéressés par la variabilité totale de l'erreur du réseau de neurones, mais par sa variabilité orientée vers les faibles erreurs. Spagnol et al. (2018) adapte le HSIC pour prendre en compte ce critère en définissant $\mathbf{Y}$, un quantile d'intérêt de l'erreur (par exemple le percentile à 10%), puis $\mathbf{z} = \mathbf{1}_{\mathbf{y} \in \mathbf{Y}}$. En calculant $HSIC(\mathbf{x}, \mathbf{z})$, la dépendance calculée est donc celle entre $\mathbf{x}$ et l'appartenance de l'erreur du réseau de neurones à son percentile à 10%.

Le HSIC de chaque hyperparamètre peut être estimé à partir d'un échantillonnage Monte Carlo. Soient $\{x_1, ..., x_{n_s}\}$ n_s échantillons d'un hyperparamètre x. Le HSIC peut être estimé à l'aide de l'estimateur $S_{x,\mathbf{Y}}$ donné par :

$$\begin{aligned} S_{x,\mathbf{Y}} = \mathbb{P}(\mathrm{z}=1)^2 \Bigg[& \frac{1}{m^2} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(x_{i,j}, x_{i,l}) \delta(z_j = 1) \delta(z_l = 1) \\ & + \frac{1}{{n_s}^2} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(x_{i,j}, x_{i,l}) \\ & - \frac{2}{n_s m} \sum_{j=1}^{n_s} \sum_{l=1}^{n_s} k(x_{i,j}, x_{i,l}) \delta(z_l = 1) \Bigg], \end{aligned} \tag{7.6}$$

avec $m = \sum_k^{n_s} \delta(z_k = 1)$ et $\delta(a) = 1$ si a est vrai et 0 sinon. Il est donc possible d'estimer les HSICs de chaque hyperparamètre après une recherche aléatoire d'hyperparamètres.

7.4.1.2 Adaptation au problème d'optimisation d'hyperparamètres

Jusqu'à présent, nous avons mentionné la possibilité de modéliser un hyperparamètre par une variable aléatoire $\mathbf{x}$ et de calculer son HSIC. La comparaison des valeurs du HSIC permettrait d'évaluer qualitativement leur importance relative par rapport à l'objectif pour le réseau de neurones d'atteindre les plus faibles erreurs. Cela serait suffisant si les hyperparamètres étaient indépendants et tous définis sur le même espace mesurable. Or, ce n'est pas le cas. En effet :

- certains sont continus, d'autres sont des entiers, catégoriels ou booléens. Il est donc compliqué de les comparer les uns avec les autres.

- Ils peuvent interagir les uns avec les autres. Ainsi, il est possible que certains hyperparamètres ne soient pas importants lorsque considérés indépendamment, mais que leurs interactions avec d'autres hyperparamètres soient importante.
- Certains ne sont pas tout le temps impliqués dans un entraînement de réseau de neurones donné (par exemple le taux de dropout (Srivastava et al., 2014) ne rentre en jeu que s'il est décidé d'appliquer du dropout au réseau de neurones). Leur présence ou non dans la configuration d'hyperparamètres dépend de la valeur d'un autre hyperparamètre (la présence du taux de dropout est conditionné à `dropout` = `Vrai`). Ces hyperparamètres sont appelés conditionnels.

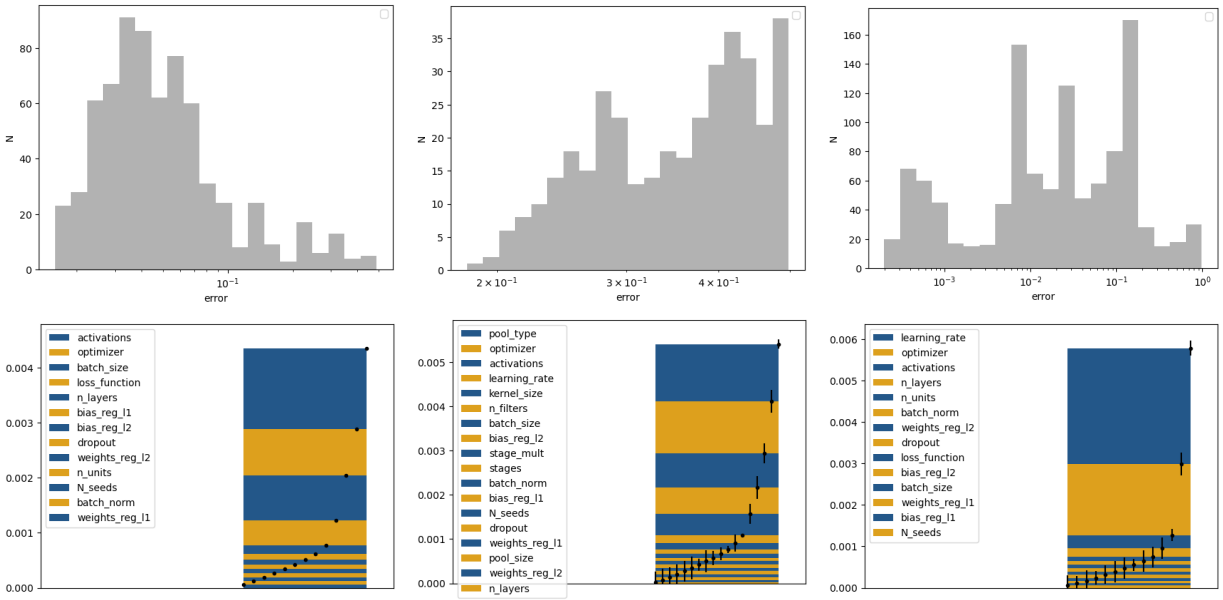
Pour le premier problème, nous proposons d'utiliser non pas les hyperparamètres eux-mêmes, mais leur image par leur fonction de densité cumulative (CDF, pour Cummulative Density Function). En effet, soit $\Phi_{\mathbf{x}}$ la CDF de $\mathbf{x}$, alors compte tenu de la définition des CDF, la variable aléatoire $\Phi_{\mathbf{x}}(\mathbf{x})$ est uniforme entre 0 et 1. En appliquant cet artifice à tous les hyperparamètres, nous les projetons dans un même espace mesurable. Ils peuvent donc y être comparés correctement.

Pour le second problème, afin de mesurer l'importance des interactions entre deux hyperparamètres $\mathbf{x}_1$ et $\mathbf{x}_2$, il est possible de calculer $HSIC(X, \mathbf{z})$, avec $X = (\mathbf{x}_1, \mathbf{x}_2)$. Lors de l'évaluation de l'importance relative des hyperparamètres, il convient donc de relever d'abord les hyperparamètres importants, c'est-à-dire à fort HSIC, puis de relever ensuite, parmi les paramètres à faible HSIC, les hyperparamètres qui sont rendus importants par leur interaction avec d'autres hyperparamètres.

Enfin, pour le troisième problème, la méthode retenue consiste à comparer le HSIC au sein de groupes d'échantillons de configurations d'hyperparamètres comportant les mêmes hyperparamètres. Un premier groupe peut être constitué, comportant les hyperparamètres qui sont impliqués dans toutes les configurations. Ces hyperparamètres sont appelés hyperparamètres principaux. L'évaluation de l'importance relative de ces hyperparamètres, basée sur les HSIC, peut se faire dans un premier temps dans ce groupe. Ensuite, le HSIC de chaque hyperparamètre conditionnel est calculé, uniquement à partir des échantillons impliquant cet hyperparamètre conditionnel. Sa valeur est finalement comparée avec celle des hyperparamètres principaux pour juger de son importance.

7.4.2 Résultats

Dans cette partie, nous appliquons l'analyse de sensibilité basée sur les HSICs à un problème de recherche d'hyperparamètres. Nous échantillonnons $n_s = 1000$ configurations différentes de n_h hyperparamètres $\{x_1, ..., x_{n_h}\}$ pour trois problèmes différents : MNIST, Cifar10 et l'approximation de la solution des équations de Bateman (décrites en **Annexe C**). Les espaces des hyperparamètres pour chacun de ces problèmes sont décrits en **Annexe B**. La figure 7.4 regroupe les histogrammes des erreurs ainsi que la comparaison des $S_{\mathbf{x}_i,\mathbf{Y}}$ pour chaque problème.



(a) MNIST (b) Cifar10 (c) Bateman

Figure 7.4: (ligne du haut) Histogramme du critère d'évaluation obtenu après l'échantillonnage aléatoire initial des configurations et (ligne du bas) comparaison des $S_{\mathbf{x}_i,\mathbf{Y}}$ pour chaque hyperparamètre principal. Pour MNIST et Cifar10, le critère d'évaluation est $1-p$, où $p \in [0,1]$ est la précision, et pour Bateman, l'erreur L_2.

Deux remarques peuvent être faites sur ces graphiques. Premièrement, le critère d'évaluation varie énormément avec les différentes configurations d'hyperparamètres. Cela justifie l'impact des hyperparamètres sur l'erreur des réseaux de neurones. Deuxièmement, les hyperparamètres les plus importants varient d'un problème à l'autre, ce qui justifie l'intérêt d'évaluer qualitativement leur importance relative pour chaque nouveau problème.

À partir des évaluations de $S_{\mathbf{x}_i,\mathbf{Y}}$, nous construisons une méthodologie d'optimisation des hyperparamètres consistant à sélectionner les paramètres les plus importants, appliquer un algorithme d'optimisation Bayesienne sur ces hyperparamètres, puis à optimiser les hyperparamètres restants. Ce faisant, la valeur des hyperparamètres les moins importants qui ont un impact sur le coût d'exécution du réseau de neurones est fixée de manière à favoriser les performances. Cette méthodologie est appelée TS-GPBO, et est comparée dans le tableau 7.4 à une optimisation Bayesienne classique sur tous les hyperparamètres (full GPBO) et à une recherche aléatoire (RS).

Ces résultats illustrent le potentiel de l'utilisation de l'analyse de sensibilité dans le cadre de la recherche d'hyperparamètres. Par exemple, pour le problème "Bateman", l'erreur obtenue avec TS-GPBO est comparable à l'erreur obtenue avec RS et full GPBO, avec un facteur 453 pour le nombre de paramètres et de FLOPs, ce qui est un résultat important pour l'appréhension du compromis précision-performances.

La méthode TS-GPBO a été construite à partir de l'analyse de sensibilité, mais n'est qu'un exemple d'application de cette technique à l'optimisation d'hyperparamètres. L'adaptation

Problème	méthode	critère d'évaluation	nb. de param.	MFLOPs
MNIST	RS	98.36	436,147	871 (×3)
-	full GPBO	**98.42** ± 0.05	10,271,367	20,534 (×67)
-	TS-GPBO	**98.42** ± 0.02	**151,306**	**307** (×1)
Cifar10	RS	81.8	99,444,880	1,832,615 (×11)
-	full GPBO	**82.73** ± 1.45	71,111,761	1,441,230 (×8)
-	TS-GPBO	79.34 ± 0.15	**9,281,258**	**178,621** (×1)
Bateman	RS	**1.99** $\times 10^{-4}$	1,259,140	2,516 (×359)
-	full GPBO	2.94 ± 0.42 $\times 10^{-4}$	1,588,215	3,173 (×453)
-	TS-GPBO	3.49 ± 0.31 $\times 10^{-4}$	**3,291**	**7** (×1)

Table 7.4: Résultats de l'optimisation d'hyperparamètres pour une recherche aléatoire (RS), pour une recherche basée sur l'optimisation Bayesienne sur tous les hyperparamètres (full GPBO) et sur l'optimisation Bayesienne en deux étapes (TS-GPBO). La moyenne ± l'écart type pour 5 répétitions est affiché pour le critère d'évaluation. Pour le nombre de paramètres et les FLOPs, seule la valeur maximum obtenue au cours des répétitions est reportée. En effet, elle illustre le pire cas qui puisse arriver pour la rapidité d'exécution, et à quel point notre méthode permet de l'éviter.

des HSICs au contexte de la recherche d'hyperparamètres présente un intérêt en tant que tel, puisqu'il s'agit d'un outil pouvant être utilisé par la communauté pour concevoir de nouvelles méthodes de recherche d'hyperparamètres basées sur l'interprétabilité.

7.5 Une analogie entre l'entraînement des réseaux de neurones et la résolution d'équations aux dérivées partielles

Dans cette partie, nous abordons la dernière étape de la méthodologie de l'apprentissage profond supervisé : l'entraînement du réseau de neurones. Cette partie, plus théorique et fondamentale, dresse un cadre pour l'entraînement des réseaux de neurones basé sur la résolution d'équations aux dérivées partielles.

L'idée des développements mathématiques de cette partie est de mettre l'équation correspondant à une itération d'un algorithme de Newton sous la forme d'une EDP de drift-diffusion. Cette EDP peut alors être résolue par des méthodes de résolution stochastique, en simulant un processus d'Itô.

Finalement, entraîner un réseau de neurones en suivant le cadre EDP revient à simuler le processus suivant :

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \frac{\Delta t}{N}\sum_{i=1}^{N}\left[\boldsymbol{\alpha}L'(\boldsymbol{x}_i,\boldsymbol{\theta}^k) + L''(\boldsymbol{x}_i,\boldsymbol{\theta}^k)\boldsymbol{\alpha}\nabla_{\boldsymbol{\theta}}^T f(\boldsymbol{x}_i,\boldsymbol{\theta}^k)(\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right] + \frac{\sqrt{\Delta t}}{N}\left[\sum_{i=1}^{N}\boldsymbol{\Sigma}(\boldsymbol{x}_i,\boldsymbol{\theta}^k,\boldsymbol{\alpha},\boldsymbol{\theta}^* - \boldsymbol{\theta}^k)\right]\mathbf{g},$$

avec $\boldsymbol{\theta}^k$ les paramètres du réseau de neurones, $f(\boldsymbol{x},\boldsymbol{\theta})$ le réseau de neurones, L la fonction de coût, $\mathbf{g}$ une Gaussienne multivariée normale, $\{\boldsymbol{x}_1,...\boldsymbol{x}_N\}$ les points d'apprentissage, et $\boldsymbol{\alpha}$, $(\boldsymbol{\theta}^* - \boldsymbol{\theta}_k)$, $\boldsymbol{\Sigma}$ des paramètres à choisir lors de l'implémentation du cadre EDP.

Nous démontrons que la descente de gradient stochastique (SGD), largement utilisée en optimisation, rentre dans le cadre EDP à condition d'appliquer des hypothèses de modélisation sur $\boldsymbol{\alpha}$, $(\boldsymbol{\theta}^* - \boldsymbol{\theta}_k)$ et $\boldsymbol{\Sigma}$. En allégeant ces hypothèses, il est possible de construire un nouvel algorithme d'optimisation, nommé PDESGD.

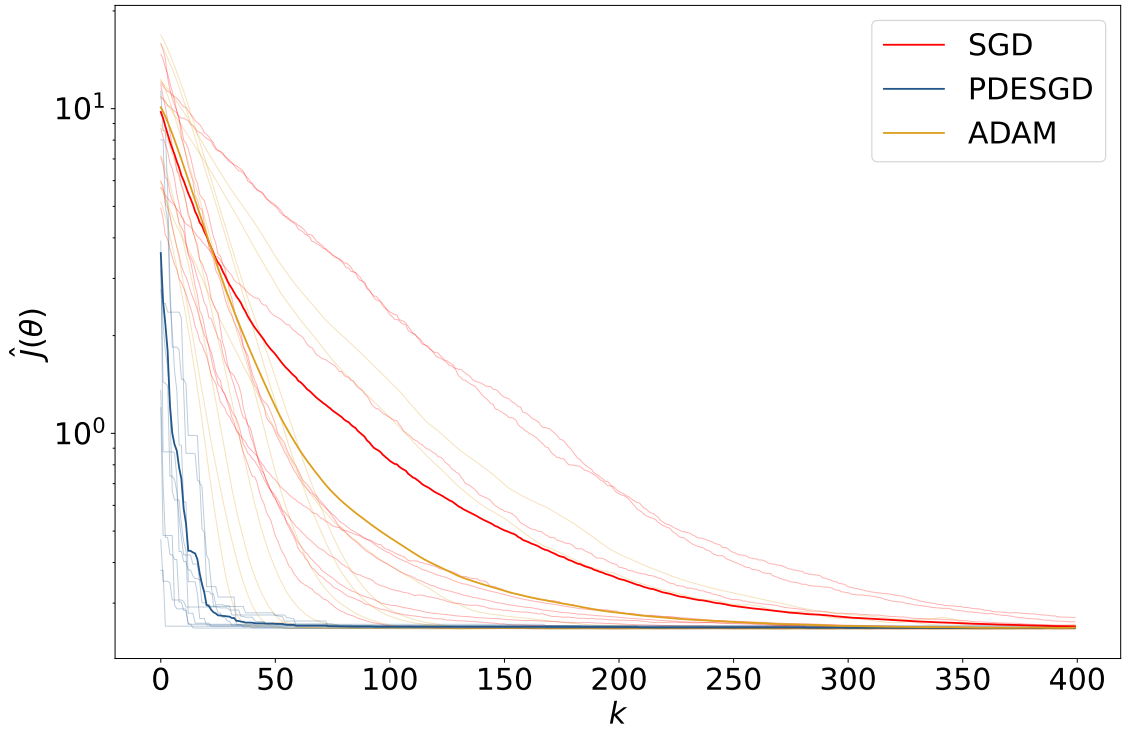


Figure 7.5: La fonction de coût intégrée J tracée en fonction du nombre d'itérations du processus d'optimisation. Les courbes sont tracées pour 40 initialisation différentes des paramètres du réseau de neurones, et la moyenne de ces courbes est tracée en gras.

Les propriétés du cadre des EDP permettent de garantir la stabilité du processus, évitant ainsi les divergences dans l'optimisation du réseau de neurones. De plus, sur un problème d'optimisation convexe mettant en jeu l'entraînement d'un réseau à deux neurones pour l'approximation d'une fonction constante, PDESGD converge plus rapidement que la SGD et qu'Adam (Kingma and Ba, 2015), un autre algorithme d'optimisation largement utilisé pour l'entraînement des réseaux de neurones. Ce résultat est illustré sur la figure 7.5.

Ces résultats sont prometteurs, car ils valident l'intérêt du cadre d'entraînement basé sur les EDPs. De plus, ce cadre présente de nombreuses perspectives. En effet, il permet de concevoir un grand nombre d'algorithmes d'optimisation par le choix de $\boldsymbol{\alpha}$, $(\boldsymbol{\theta}^* - \boldsymbol{\theta}_k)$ et $\boldsymbol{\Sigma}$. Tout en incluant cette composante heuristique, il bénéficie cependant de toutes les propriétés de la théorie de résolution des EDPs.

7.6 Application à la construction d'un code de simulation hybride

Après avoir étudié les différentes étapes de la méthodologie de l'apprentissage profond supervisé, nous appliquons cette méthodologie à un cas concret consistant à accélérer un code de simulation numérique hydrodynamique. Cette simulation étudie l'écoulement d'un fluide autours d'un objet, illustré en figure 7.6. L'application d'un tel code peut être par exemple la rentrée d'un satellite ou d'une sonde dans l'atmosphère. Ce code est couplé à un outil, Mutation++ (Scoggins et al., 2020), modélisant l'équilibre entre les espèces chimiques présentes dans le fluide étudié.

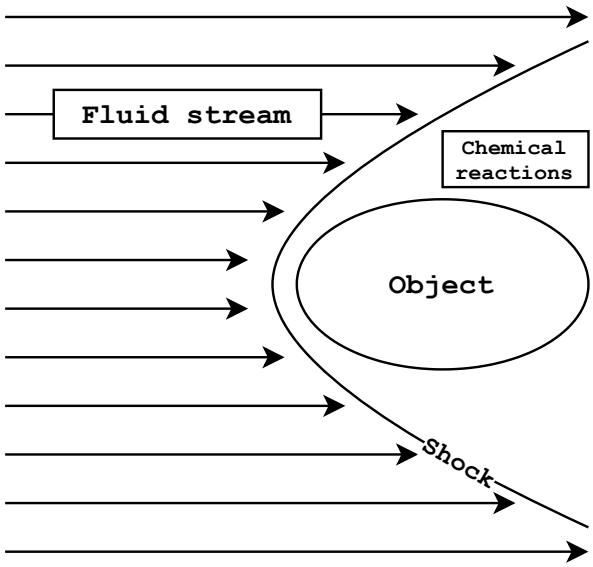


Figure 7.6: Illustration de la simulation.

7.6.1 Description du code

Le code global suit une structure de la forme suivante :

$$\begin{cases} F_1(\boldsymbol{\eta}, \boldsymbol{u}, \boldsymbol{\alpha}) = 0, & \text{(7.7a)} \\ F_2(\boldsymbol{\eta}, \boldsymbol{u}, \boldsymbol{\alpha}) = 0, & \text{(7.7b)} \end{cases}$$

où F_1 correspond aux équations d'Euler, F_2 correspond à la minimisation de l'enthalpie libre, $\boldsymbol{u}$ correspond aux variables conservatives de l'écoulement (densité, pression, vitesse), η correspond aux fractions molaires des espèces chimiques en jeu et α aux constantes utilisées dans la simulation. L'opérateur F_1 modélise l'écoulement du fluide, et n'est pas coûteux lorsqu'exécuté seul. Cependant, sa résolution itérative implique l'appel à Mutation ++ à chaque itération, pour résoudre F_2. Au total, l'exécution du code est donc coûteuse à cause des appels à Mutation++ : même si un appel isolé à Mutation++ n'est pas couteux, la répétition de cet appel en chaque maille et à chaque itération représente la majeure partie du temps d'exécution. Il y a donc beaucoup à gagner à tenter de réduire le temps d'exécution imputable à Mutation++.

Notons avant de poursuivre que la structure décrite dans l'équation 7.7 n'est pas spécifique à ce code, mais peut se retrouver dans de nombreuses simulations. Par exemple, en neutronique, F_1 peut être l'équation de Boltzman et F_2 l'équation de Bateman (Bernède and Poëtte, 2018; Dufek et al., 2013).

Afin de construire le code hybride, nous entraînons donc un réseau de neurones pour l'approximation de Mutation++. Une fois le réseau de neurones entraîné, il est possible de le coupler avec le code en l'exportant et en l'utilisant à l'aide de l'API C de Tensorflow (Abadi et al., 2015).

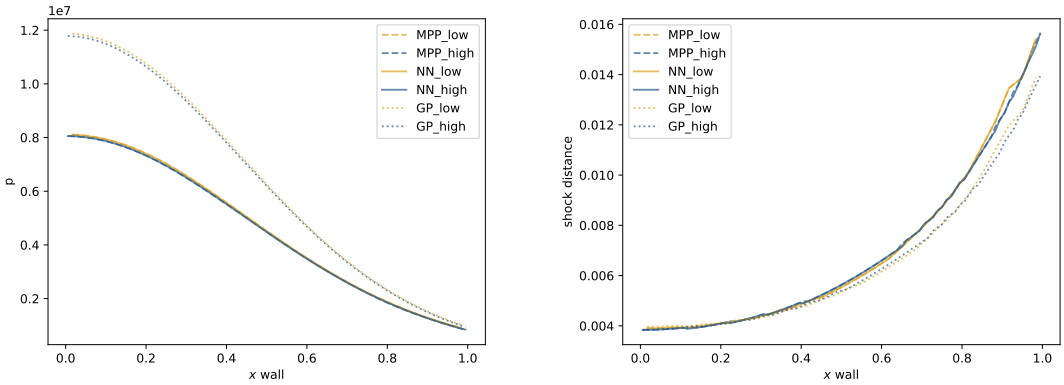
7.6.2 Résultats

Dans cette partie, nous comparons les résultats du code hybride (NN) avec ceux du code original (MPP). Nous comparons également ces codes avec une résolution de F_1 ne faisant pas appel à Mutation++ en utilisant l'hypothèse des gaz parfaits (PG). Le réseau de neurones utilisé pour construire NN est entraîné sur une base de données échantillonnée uniformément sur le sous espace d'intérêt et ses hyperparamètres sont choisis à l'aide d'une recherche aléatoire. Sur la Figure 7.7 sont tracés le profil de pression à la paroi ainsi que la distance au choc, pour NN, MPP et PG exécutés sur un maillage de taille 30×100.

Cette figure illustre deux points. Premièrement, la différence visuelle entre PG d'une part et MPP et NN d'autre part illustre l'importance de prendre en compte la chimie dans la simulation du phénomène. Deuxièmement, les prédictions de MPP et NN ne sont presque pas distinguable visuellement.

Afin d'illustrer les bénéfices tirés des méthodologies présentées dans les sections précédentes, nous entraînons le réseau de neurones pour l'approximation de Mutation++ en utilisant la méthodologie TS-GPBO, basée sur les HSICs avec la possibilité d'utiliser un échantillonnage de type TBS. Les résultats obtenus sont rassemblés dans le tableau 7.5, où sont évaluées les erreurs L_2 et L_∞ de la prédiction sur le profil de pression et la distance au choc. Ces erreurs sont calculées par rapport à un profil obtenu avec un maillage de taille 90×100.

Ces résultats illustrent à la fois l'intérêt de la construction d'un code hybride et l'intérêt de la



(a) (b)

Figure 7.7: **(a)** Profil de pression **(b)** Distance au choc

	MPP	MPP + NN	NN		PG
Sélection d'hyp.	TS-GPBO	-	RS	TS-GPBO	-
Temps (s)	4090	383	531	220	81
Amélioration (×)	-	10.5	8.9	18.7	56.5
Pression					
L^2	1.09×10^{-4}	-	1.06×10^{-4}	1.24×10^{-4}	8.38×10^{-2}
L^∞	1.48×10^{-2}	-	1.53×10^{-2}	1.57×10^{-2}	4.75×10^{-1}
Dist. choc					
L^2	7.40×10^{-5}	-	6.43×10^{-5}	7.53×10^{-5}	1.77×10^{-3}
L^∞	2.80×10^{-2}	-	2.79×10^{-2}	2.79×10^{-2}	1.02×10^{-1}

Table 7.5: Temps d'exécution et erreurs normalisées pour les différents types de code

recherche d'hyperparamètres basée sur l'analyse HSIC. En effet, l'erreur obtenue avec NN est équivalente à celle obtenue avec MPP, pour un facteur d'accélération pouvant aller jusqu'à 18.7. Lorsque la prédiction de NN est donnée en initialisation de MPP (dénoté NN+MPP), le facteur d'accélération est de 10.5 pour le même résultat que MPP. De plus, utiliser TS-GPBO permet de réduire le temps total d'exécution par deux sans affecter significativement l'erreur de prédiction. Il est à noter que le meilleur réseau de neurones obtenu avec RS utilise un échantillonnage TBS.

Cette application démontre le potentiel de la construction de codes hybrides pour la simulation numérique. L'erreur provenant de l'approximation de Mutation++ par un réseau de neurones est limitée, ce qui joue en faveur de leur fiabilité. Le gain de temps induit permettrait d'accélérer les délais de production et les études paramétriques liées à la conception de systèmes. Il pourrait également permettre d'évaluer rapidement l'intérêt de prendre en compte certaines physiques dans un code, accélérant par là les processus de décision liés au

développement du code.

7.7 Conclusion

L'objectif premier de cette thèse était d'étudier l'apprentissage profond supervisé dans l'optique de l'utiliser pour accélérer des codes de simulation numérique. Par conséquent, nous avons étudié cette technique à travers le prisme de l'analyse numérique et de l'analyse d'incertitudes. Cela nous a mené à des contributions ayant un intérêt à la fois pour le domaine de l'apprentissage automatique et pour celui de la simulation numérique: TBS et VBSW, des méthodes pour mieux construire sa base d'entraînement; l'utilisation du HSIC pour sélectionner les hyperparamètres du réseau de neurones en tenant compte du compromis précision-performances lié à notre premier objectif; l'utilisation de la théorie des EDP pour optimiser les réseaux de neurones; et enfin l'implémentation d'un réseau de neurones dans un code de simulation, menant à un code de simulation hybride, plus rapide que le code original d'un facteur 10 à 20 sans sacrifice de garantie.

Chapter 8

Conclusions

The first goal of this work was to study supervised deep learning as a tool to accelerate numerical simulations. Hence, we carefully investigated the supervised learning methodology through the prism of numerical and uncertainty analysis, which are strongly related to numerical simulations. It led to contributions of interest for both machine learning and scientific computing. Finally, we accelerated a CFD simulation code by applying this methodology and constructing a hybrid numerical simulation based on both deep learning and the original simulation code.

8.1 Contribution to the methodology of supervised deep learning

Supervised deep learning can be divided into three steps: the construction of the training database; the choice of the neural network's hyperparameters; and its training, or learning, using optimization algorithms; so do our three first contributions.

8.1.1 Training distribution and local variations

We first studied the training distribution of the neural network. By doing so, we explored a practical and classical question that naturally arises when performing surrogate modeling for approximating simulation codes: how to construct the training set? This contribution is based on the observation that neural networks are more efficient when more data are sampled where the function to be learnt is steeper.

We motivated this observation with an illustrative generalization bound and constructed a methodology, Taylor Based Sampling (TBS), to verify it empirically. TBS reduced the

L_2 and L_∞ errors for the approximation of Runge function, hyperbolic tangent, and the solution of Bateman equations. We then derived Variance Based Sample Weighting (VBSW) from alleviating the limitations of TBS, which is not applicable for general machine learning tasks. We validated VBSW on several applications, such as glue benchmark with bert for text classification and regression and Cifar10 with ResNet for image classification.

Although VBSW uses theoretically justified approximations concerning TBS, the actual effect of these approximations should be more thoroughly investigated. For instance, we could further study the impact of not using $p_{\mathbf{x}}$, the data distribution, in the weights definitions; the convergence of the estimator of $Df_{\boldsymbol{\epsilon}}^2$, and in which context it is adequately approximated; and a more generic derivative-based generalization bound. In addition, VBSW demonstrated intriguing behaviors, like its impressive synergy with TCL (Hacohen and Weinshall, 2019), which would deserve more attention.

The results obtained in this chapter are impactful for machine learning in numerical simulations and machine learning in general. For the former, it can be seen as a new way of constructing designs of experiments, using the distribution $d\mathbb{P}_{\bar{\mathbf{x}}}$ underlying TBS and VBSW, which involves the derivatives of the function to approximate (f). For the latter, VBSW validates an original view of the learning problem based on the variations of f, which would be worth investigating in future works.

8.1.2 Hyperparameter optimization using goal-oriented sensitivity analysis

Then, we investigated the problem of hyperparameter optimization. This step of supervised deep learning carries high stakes for numerical simulations because hyperparameters can impact both the accuracy and the cost efficiency of neural networks.

We adapted HSIC based goal-oriented sensitivity analysis, an approach widely used in numerical and uncertainty analysis, to the context of hyperparameter analysis. Indeed, hyperparameters can be of different natures, like continuous, discrete, categorical, or boolean, and have non-trivial relations, like conditionality or interactions. We designed a two-step optimization methodology, TS-GPBO, based on feature selection that tackles the performance-accuracy trade-off by identifying hyperparameters that impact performance but not accuracy. In our experiments, TS-GPBO yields neural networks with at least ten times less parameters than those obtained with traditional Bayesian Optimization.

Here, we chose HSIC as a dependence measure. Studying other dependence measures to test the robustness of the qualitative insights earned by HSIC and characterize them more precisely could be interesting. We could also study the effect of using different kernels to obtain other properties (like orthogonality, as done in da Veiga (2021)). In addition, the idea to transform hyperparameters distribution into uniform could be tested for other hyperparameter optimization algorithms.

Finally, though TS-GPBO and other presented methodologies can be taken as contributions by themselves, they should also be understood as demonstrations that HSIC based goal-oriented sensitivity analysis is interesting and valuable for hyperparameter optimization. In the end, an essential outcome of this work was to make an insightful tool, HSIC, available to the community in the context of hyperparameter optimization.

8.1.3 A view of learning from the Partial Differential Equation theory

We then focused on the optimization step of the supervised deep learning process. We achieved valuable results by revisiting the optimization at play in machine learning using stochastic PDE theory. First, we defined a PDE framework that links PDE and optimization algorithms. This framework is subject to modeling choices (on $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}^* - \boldsymbol{\theta}$) that lead to algorithms characterized by exploration as well as optimization.

To illustrate the benefits of this framework, we constructed PDESGD, a new optimization algorithm. We defined an adaptative learning rate to impose stability of the optimization that allows efficient and stable exploration of the parameter space. We also compared PDESGD with SGD and Adam in the convex regime, exhibiting faster convergence in the presented test case.

The work presented in this chapter is mainly theoretical. The experiments with PDESGD are convincing, but the test case is too simplistic to conclude its relevance for large-scale deep learning problems. In addition, the analysis only holds for neural networks with scalar output. Such a study would be in the continuity of this work: it is the closest perspective.

Then, more experimental work could help to elaborate on the choices for α and $\boldsymbol{\theta}^* - \boldsymbol{\theta}$. To that end, it should be useful to study the properties of the solved PDE for each different choice. Finally, the PDE framework relies on the assumption that $\hat{f} \sim 0$. There is an interest in assessing this assumption to improve the optimization. The report of Poëtte et al. (2021) is an attempt to evaluate this hypothesis using transport PDE theory.

8.2 Application of the methodology to the construction of a hybrid CFD numerical simulation

We then applied supervised deep learning to approximate chemical equilibrium, initially computed by Mutation++ (Scoggins et al., 2020), in a CFD code. We embedded the obtained neural networks in the code and brought a hybrid code significantly faster than the original one.

We also demonstrated how prediction accuracy could be guaranteed along with this acceleration.

First, by initializing the original code with the prediction of the hybrid code, one can obtain a factor 10 speed-up for the exact prediction. Second, we showed that neural network approximation error is negligible compared to classical errors inherent to numerical simulations (uncertainty, discretization, and model error). Hence, we tend to claim that the hybrid code alone, with a time improvement factor of 21, could be safely used.

The methodology for approximating Mutation++ can only be used for test cases that involve a fixed list of species. This constraint is an obstacle towards a general usage of neural networks to approximate Mutation++ in concerned simulation codes. Indeed, it requires constructing a new training database for each different chemical setting. This problem should be addressed because building a neural network approximating Mutation++ for a large number of other test cases would significantly improve the versatility of the approach and ease its broader application.

Finally, the formulation of the resolution of coupled equations is quite general and encompasses many phenomena in addition to our test case. Hence, it would be valuable to test the approach to simulate other phenomena to challenge its efficiency further.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Kim Albertsson, Piero Altoe, Dustin Anderson, Michael Andrews, Juan Pedro Araque Espinosa, Adam Aurisano, Laurent Basara, Adrian Bevan, Wahid Bhimji, Daniele Bonacorsi, Paolo Calafiura, Mario Campanelli, Louis Capps, Federico Carminati, Stefano Carrazza, Taylor Childers, Elias Coniavitis, Kyle Cranmer, Claire David, Douglas Davis, Javier Duarte, Martin Erdmann, Jonas Eschle, Amir Farbin, Matthew Feickert, Nuno Filipe Castro, Conor Fitzpatrick, Michele Floris, Alessandra Forti, Jordi Garra-Tico, Jochen Gemmler, Maria Girone, Paul Glaysher, Sergei Gleyzer, Vladimir Gligorov, Tobias Golling, Jonas Graw, Lindsey Gray, Dick Greenwood, Thomas Hacker, John Harvey, Benedikt Hegner, Lukas Heinrich, Ben Hooberman, Johannes Junggeburth, Michael Kagan, Meghan Kane, Konstantin Kanishchev, Przemyslaw Karpinski, Zahari Kassabov, Gaurav Kaul, Dorian Kcira, Thomas Keck, Alexei Klimentov, Jim Kowalkowski, Luke Kreczko, Alexander Kurepin, Rob Kutschke, Valentin Kuznetsov, Nicolas Köhler, Igor Lakomov, Kevin Lannon, Mario Lassnig, Antonio Limosani, Gilles Louppe, Aashrita Mangu, Pere Mato, Helge Meinhard, Dario Menasce, Lorenzo Moneta, Seth Moortgat, Meenakshi Narain, Mark Neubauer, Harvey Newman, Hans Pabst, Michela Paganini, Manfred Paulini, Gabriel Perdue, Uzziel Perez, Attilio Picazio, Jim Pivarski, Harrison Prosper, Fernanda Psihas, Alexander Radovic, Ryan Reece, Aurelius Rinkevicius, Eduardo Rodrigues, Jamal Rorie, David Rousseau, Aaron Sauers, Steven Schramm, Ariel Schwartzman, Horst Severini, Paul Seyfert, Filip Siroky, Konstantin Skazytkin, Mike Sokoloff, Graeme Stewart, Bob Stienen, Ian Stockdale, Giles Strong, Savannah Thais, Karen Tomko, Eli Upfal, Emanuele Usai, Andrey Ustyuzhanin, Martin Vala, Sofia Vallecorsa, Justin Vasel, Mauro Verzetti, Xavier

Vilasis-Cardona, Jean-Roch Vlimant, Ilija Vukotic, Sean-Jiun Wang, Gordon Watts, Michael Williams, Wenjing Wu, Stefan Wunsch, and Omar Zapata. Machine learning in high energy physics community white paper. *Journal of Physics: Conference Series*, 1085:022008, sep 2018.

Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 254–263, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

Javier Arroyo, Samer Hassan, Celia Gutierrez, and Juan Pavon. Re-thinking simulation: a methodological approach for the application of data mining in agent-based modelling. *Computational and Mathematical Organization Theory*, 16(4):416–435, Dec 2010.

Peter Auer, Mark Herbster, and Manfred K. K Warmuth. Exponentially many local minima for single neurons. In D. Touretzky, M. C. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8. MIT Press, 1996.

Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, Karen Willcox, and Steven Lee. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. 2 2019.

Ruth E. Baker, Jose-Maria Peña, Jayaratnam Jayamohan, and Antoine Jérusalem. Mechanistic models versus machine learning, a fight worth fighting for the biological community? *Biology Letters*, 14(5):20170660, 2018.

Andrew R. Barron. Approximation and estimation bounds for artificial neural networks. *Machine Learning*, 14(1):115–133, Jan 1994.

Peter L. Bartlett, Vitaly Maiorov, and Ron Meir. Almost linear vc dimension bounds for piecewise polynomial networks. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, NIPS'98, page 190–196, Cambridge, MA, USA, 1998. MIT Press.

Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6240–6249. Curran Associates, Inc., 2017.

Peter L. Bartlett, Nick Harvey, Christopher Liaw, and Abbas Mehrabian. Nearly-tight vc-dimension and pseudodimension bounds for piecewise linear neural networks. *Journal of Machine Learning Research*, 20(63):1–17, 2019.

Andrea D. Beck, Jonas Zeifang, Anna Schwarz, and David G. Flad. A neural network based shock detection and localization approach for discontinuous galerkin methods. *Journal of Computational Physics*, 423:109824, 2020.

Sue Becker and Yann L. Cun. Improving the convergence of back-propagation learning with second order methods. In David S. Touretzky, Geoffrey E. Hinton, and Terrence J. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37. San Francisco, CA: Morgan Kaufmann, 1989.

Jörg Behler and Michele Parrinello. Generalized neural-network representation of high-dimensional potential-energy surfaces. *Phys. Rev. Lett.*, 98:146401, Apr 2007.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, March 1994.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009a. ACM.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009b. ACM.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

Tristan Bereau, Robert A. DiStasio, Alexandre Tkatchenko, and O. Anatole von Lilienfeld. Non-covalent interactions across organic and biological subsets of chemical space: Physics-based potentials parametrized from machine learning. *The Journal of Chemical Physics*, 148(24):241706, June 2018.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012.

James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2546–2554. Curran Associates, Inc., 2011.

Adrien Bernède and Gaël Poëtte. An unsplit monte-carlo solver for the resolution of the linear boltzmann equation coupled to (stiff) bateman equations. *Journal of Computational Physics*, 354:211–241, 02 2018.

M. Bisi and L. Desvillettes. From reactive boltzmann equations to reaction–diffusion systems. *Journal of Statistical Physics*, 124(2):881–912, Aug 2006.

Géraud Blatman and Bruno Sudret. Sparse polynomial chaos expansions and adaptive stochastic finite elements using a regression approach. *Comptes Rendus Mécanique*, 336(6): 518–523, June 2008.

Bastian Bohn, Jochen Garcke, Rodrigo Iza-Teran, Alexander Paprotny, Benjamin Peherstorfer, Ulf Schepsmeier, and Clemens-August Thole. Analysis of car crash simulation data with nonlinear machine learning methods. *Procedia Computer Science*, 18:621–630, 2013. 2013 International Conference on Computational Science.

E. Borgonovo. A new uncertainty importance measure. *Reliability Engineering & System Safety*, 92(6):771 – 784, 2007.

Nick Bostrom. Are you living in a computer simulation? *Philosophical Quarterly*, 53(211): 243–255, 2003.

Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

Alexandre Bourriaud, Raphaël Loubère, and Rodolphe Turpault. A Priori Neural Networks Versus A Posteriori MOOD Loop: A High Accurate 1D FV Scheme Testing Bed. *Journal of Scientific Computing*, 84(2):31, August 2020.

Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In Nieves R. Brisaboa, Oscar Pedreira, and Pavel Zezula, editors, *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer, 2013.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Haim Brezis. *Function Analysis, Sobolev Spaces and Partial Differential Equations*. 01 2010.

Felix Brockherde, Leslie Vogt, Li Li, Mark E. Tuckerman, Kieron Burke, and Klaus-Robert Müller. Bypassing the Kohn-Sham equations with machine learning. *Nature Communications*, 8(1):872, December 2017.

Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

Richard H. Byrd, Gillian M. Chin, Will Neveitt, and Jorge Nocedal. On the Use of Stochastic Hessian Information in Optimization Methods for Machine Learning. *SIAM Journal on Optimization*, 21(3):977–995, July 2011.

Shengze Cai, Zhiping Mao, Zhicheng Wang, Minglang Yin, and George Em Karniadakis. Physics-informed neural networks (pinns) for fluid mechanics: A review, 2021.

José Antonio Carrillo and Mattia Zanella. Monte carlo gpc methods for diffusive kinetic flocking models with uncertainties. *Vietnam Journal of Mathematics*, 47(4):931–954, 2019.

Rich Caruana, Steve Lawrence, and C. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2001.

John I. Castor. *Radiation Hydrodynamics*. Cambridge University Press, 2004.

Haw-Shiuan Chang, Erik Learned-Miller, and Andrew McCallum. Active bias: Training more accurate neural networks by emphasizing high variance samples. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1002–1012. Curran Associates, Inc., 2017.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, 2014. Association for Computational Linguistics.

François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

J.-F. Clouët and G. Samba. Asymptotic diffusion limit of the symbolic Monte-Carlo method for the transport equation. *Journal of Computational Physics*, 195(1):293–319, March 2004.

David A. Cohn. Neural Network Exploration Using Optimal Experiment Design. *Neural Networks*, 9(6):1071–1083, August 1996.

P.M. Congedo, C. Corre, and J.-M. Martinez. Shape optimization of an airfoil in a bzt flow with multiple-source uncertainties. *Computer Methods in Applied Mechanics and Engineering*, 200(1):216–232, 2011.

Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust-Region Methods*. Society for Industrial and Applied Mathematics, USA, 2000.

I. Csizar. Information-type measures of difference of probability distributions and indirect observation. *Studia Scientiarum Mathematicarum Hungarica*, 2:229–318, 1967.

Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

Lucor D., Xiu D., Su C.H., and Karniadakis G.E. Predictability and uncertainty in CFD. *International Journal for Numerical Methods in Fluids*, 43:483–505, 2003.

S. Da Veiga, F. Wahl, and F. Gamboa. Local polynomial estimation for sensitivity analysis on models with correlated inputs. *Technometrics*, 51:452 – 463, 2009.

Sébastien Da Veiga. Global sensitivity analysis with dependence measures. *Journal of Statistical Computation and Simulation*, 85, 11 2013.

Sébastien da Veiga. Kernel-based anova decomposition and shapley effects – application to global sensitivity analysis, 2021.

Florian Danvin, Marina Olazabal, and Fabio Pinna. *Laminar to turbulent transition prediction in hypersonic flows with neural networks committee*. 2021.

M S Day and J B Bell. Numerical simulation of laminar reacting flows with complex chemistry. *Combustion Theory and Modelling*, 4(4):535–556, December 2000.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

S. Doerr and G. De Fabritiis. On-the-fly learning and sampling of ligand binding by high-throughput molecular simulations. *Journal of Chemical Theory and Computation*, 10(5): 2064–2069, May 2014.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Jan Dufek, Dan Kotlyar, and Eugene Shwageraus. The stochastic implicit euler method – a stable coupling scheme for monte carlo burnup calculations. *Annals of Nuclear Energy*, 60: 295 – 300, 10 2013.

A.E. Eiben and M. Schoenauer. Evolutionary computing. *Information Processing Letters*, 82 (1):1–6, 2002. Evolutionary Computation.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

J. Feng, Q. Teng, X. He, and X. Wu. Accelerating multi-point statistics reconstruction method for porous media via deep learning. *Acta Materialia*, 159:296–308, 2018.

Krzysztof J. Fidkowski and Guodong Chen. Metric-based, goal-oriented mesh adaptation using machine learning. *Journal of Computational Physics*, 426:109957, 2021.

Jean-Claude Fort, Thierry Klein, and Nabil Rachdi. New sensitivity analysis subordinated to a contrast. *Communications in Statistics - Theory and Methods*, 45(15):4349–4364, 2016.

Jerome H. Friedman. Multivariate adaptive regression splines. *Ann. Statist*, 1991.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

Kenji Fukumizu, Arthur Gretton, Gert R. Lanckriet, Bernhard Schölkopf, and Bharath K. Sriperumbudur. Kernel choice and classifiability for rkhs embeddings of probability distributions. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1750–1758. Curran Associates, Inc., 2009.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, Apr 1980.

Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

Yarin Gal, Riashat Islam, and Zoubin Ghahramani. Deep bayesian active learning with image data. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 1183–1192. JMLR.org, 2017.

Saul B. Gelfand and Sanjoy K. Mitter. Recursive Stochastic Algorithms for Global Optimization in $\mathbb{R}^d $. *SIAM Journal on Control and Optimization*, 29(5):999–1018, September 1991.

Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403 – 434, 1976.

Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins studies in the mathematical sciences. The Johns Hopkins University Press, Baltimore, fourth edition edition, 2013. OCLC: ocn824733531.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael Cree. Regularisation of neural networks by enforcing lipschitz continuity. 04 2018.

Robert B. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall/CRC, Boca Raton, Florida, 2020. `http://bobby.gramacy.com/surrogates/`.

Arthur Gretton, Olivier Bousquet, Alex Smola, and Bernhard Schölkopf. Measuring statistical dependence with hilbert-schmidt norms. In *Proceedings of the 16th International Conference on Algorithmic Learning Theory*, ALT'05, page 63–77, Berlin, Heidelberg, 2005. Springer-Verlag.

Arthur Gretton, Karsten Borgwardt, Malte Rasch, Bernhard Schölkopf, and Alex J. Smola. A kernel method for the two-sample-problem. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 513–520. MIT Press, 2007.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Xiaoxiao Guo, Wei Li, and Francesco Iorio. Convolutional neural networks for steady flow approximation. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 481–490, New York, NY, USA, 2016. Association for Computing Machinery.

Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2535–2544. PMLR, 09–15 Jun 2019.

Zhongqing Han, Rahul, and Suvranu De. A deep learning-based hybrid approach for the solution of multiphysics problems in electrosurgery. *Computer Methods in Applied Mechanics and Engineering*, 357:112603, December 2019.

Haowei He, Gao Huang, and Yang Yuan. Asymmetric valleys: Beyond sharp and flat local minima. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9 (8):1735–1780, 1997.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1729–1739, Red Hook, NY, USA, 2017. Curran Associates Inc.

J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

S. Hosder, R. W. Walters, and R. Perez. A Non Intrusive Polynomial Chaos Method for Uncertainty Propagation in CFD Simulations. *44th AIAA Aerospace Sciences Meeting and Exhibit*, AIAA 2006-891, 2006.

Jianguo Huang, Haoqin Wang, and Haizhao Yang. Int-Deep: A deep learning initialized iterative method for nonlinear problems. *Journal of Computational Physics*, 419:109675, October 2020.

Jorge E. Hurtado. Neural networks in stochastic mechanics. *Archives of Computational Methods in Engineering*, 8(3):303–342, Sep 2001.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In Ricardo Silva, Amir Globerson, and Amir Globerson, editors, *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, 34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018, pages 876–885. Association For Uncertainty in Artificial Intelligence (AUAI), 2018a. Funding Information: Acknowledgements. This work was supported by NSF IIS-1563887, Samsung Research, Samsung Electronics and Russian Science Foundation grant 17-11-01027. We also thank Vadim Bereznyuk for helpful comments. Funding Information: This work was supported by NSF IIS-1563887, Samsung Research, Samsung Electronics and Russian Science Foundation grant 17-11-01027. We also thank Vadim Bereznyuk for helpful comments. Publisher Copyright: © 34th Conference on Uncertainty in Artificial Intelligence 2018. All rights reserved.; 34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018 ; Conference date: 06-08-2018 Through 10-08-2018.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In Ricardo Silva, Amir Globerson, and Amir Globerson, editors, *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018*, 34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018, pages 876–885. Association For Uncertainty in Artificial Intelligence (AUAI), 2018b. Funding Information: Acknowledgements. This work was supported by NSF IIS-1563887, Samsung Research, Samsung Electronics and Russian Science Foundation grant 17-11-01027. We also thank Vadim Bereznyuk for helpful comments. Funding Information: This work was supported by NSF IIS-1563887, Samsung Research, Samsung Electronics and Russian Science Foundation grant 17-11-01027. We also thank Vadim Bereznyuk for helpful comments. Publisher Copyright: © 34th Conference on Uncertainty in Artificial Intelligence 2018. All rights reserved.; 34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018 ; Conference date: 06-08-2018 Through 10-08-2018.

Daniel Jakubovitz, Raja Giryes, and Miguel R. D. Rodrigues. Generalization error in deep learning. *CoRR*, abs/1808.01174, 2018.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In Arthur Gretton and Christian C. Robert, editors, *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 09–11 May 2016. PMLR.

Lu Jiang, Deyu Meng, Qian Zhao, Shiguang Shan, and Alexander G. Hauptmann. Self-paced

curriculum learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 2694–2700. AAAI Press, 2015.

Tang Jie and Pieter Abbeel. On a connection between importance sampling and the likelihood ratio policy gradient. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1000–1008. Curran Associates, Inc., 2010.

M.E. Johnson, L.M. Moore, and D. Ylvisaker. Minimax and maximin distance designs. *Journal of Statistical Planning and Inference*, 26(2):131–148, October 1990.

Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *J. of Global Optimization*, 13(4):455–492, December 1998a.

Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998b.

Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2020–2029, Red Hook, NY, USA, 2018. Curran Associates Inc.

Angelos Katharopoulos and François Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

Marc C. Kennedy and Anthony O'Hagan. Bayesian calibration of computer models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(3):425–464, 2001.

Nitish Shirish Keskar, Jorge Nocedal, Ping Tak Peter Tang, Dheevatsa Mudigere, and Mikhail Smelyanskiy. On large-batch training for deep learning: Generalization gap and sharp minima. 2017. 5th International Conference on Learning Representations, ICLR 2017 ; Conference date: 24-04-2017 Through 26-04-2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

G. Kluth, K. D. Humbird, B. K. Spears, J. L. Peterson, H. A. Scott, M. V. Patel, J. Koning, M. Marinak, L. Divol, and C. V. Young. Deep learning for nlte spectral opacities. *Physics of Plasmas*, 27(5):052707, 2020.

Gilles Kluth and Bruno Després. 2d finite volume lagrangian scheme in hyperelasticity and finite plasticity. In *Numerical Mathematics and Advanced Applications 2009*, pages 489–496. Springer, 2010.

Gilles Kluth, Kelli Humbird, Brian Spears, Howard Scott, Mehul Patel, Luc Peterson, Joe Koning, Marty Marinak, Laurent Divol, and Chris Young. Deep Learning for Non-Local Thermodynamic Equilibrium in hydrocodes for ICF. In *APS Division of Plasma Physics Meeting Abstracts*, volume 2019 of *APS Meeting Abstracts*, page BO5.009, January 2019.

Ksenia Konyushkova, Raphael Sznitman, and Pascal Fua. Learning active learning from data. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4225–4235. Curran Associates, Inc., 2017.

Slawomir Koziel and Leifur Leifsson. Knowledge-based airfoil shape optimization using space mapping. In *30th AIAA Applied Aerodynamics Conference*, 2012.

Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

Anders Krogh and John Hertz. A simple weight decay can improve generalization. In J. Moody, S. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4. Morgan-Kaufmann, 1992.

J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, March 1964.

M. P. Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1189–1197. Curran Associates, Inc., 2010.

Yongchan Kwon, Joong-Ho Won, Beom Joon Kim, and Myunghee Cho Paik. Uncertainty quantification using bayesian neural networks in classification: Application to biomedical image segmentation. *Computational Statistics & Data Analysis*, 142:106816, 2020.

William Albert Lahoz, Boris Khattatov, and Richard Ménard, editors. *Data assimilation: making sense of observations*. Springer, Heidleberg, New York, 2010. OCLC: ocn499067426.

Jouko Lampinen and Aki Vehtari. Bayesian approach for neural networks—review and case studies. *Neural Networks*, 14(3):257–274, 2001.

Hunter Lang, Lin Xiao, and Pengchuan Zhang. Using statistics to automate stochastic optimization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

O. Le Maitre, M. Reagan, H. Najm, R. Ghanem, and O. Knio. A Stochastic Projection Method for Fluid Flow. II. Random Process. *J. Comp. Phys.*, 181:9–44, 2002.

O. P. Le Maître and O. M. Knio. Uncertainty Propagation using Wiener-Haar Expansions. *J. Comp. Phys.*, 197:28–57, 2004.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018a.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018b.

Tongliang Liu and Dacheng Tao. Classification with noisy labels by importance reweighting. *IEEE Trans. Pattern Anal. Mach. Intell.*, 38(3):447–461, March 2016.

Raphaël Loubère, Michael Dumbser, and Steven Diot. A New Family of High Order Unstructured MOOD and ADER Finite Volume Schemes for Multidimensional Systems of Hyperbolic Conservation Laws. *Communications in Computational Physics*, 16(3):718–763, September 2014.

Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, Mar 2021.

Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 6231–6239. Curran Associates, Inc., 2017.

D. Lucor, C. Enaux, H. Jourdren, and P. Sagaut. Stochastic design optimization: Application to reacting flows. *Computer Methods in Applied Mechanics and Engineering*, 196(49):5047 – 5062, 2007a.

D. Lucor, C. Enaux, H. Jourdren, and P. Sagaut. Multi-Physics Stochastic Design Optimization: Application to Reacting Flows and Detonation. *Comp. Meth. Appl. Mech. Eng.*, 196: 5047–5062, 2007b.

D. Lucor, J. Meyers, and P. Sagaut. Sensitivity Analysis of LES to Subgrid-Scale-Model Parametric Uncertainty using Polynomial Chaos. *J. Fluid Mech.*, 585:255–279, 2007c.

David J. C. MacKay. Information-Based Objective Functions for Active Data Selection. *Neural Computation*, 4(4):590–604, July 1992.

Pierre-Henri Maire, Rémi Abgrall, Jérôme Breil, and Jean Ovadia. A cell-centered lagrangian scheme for two-dimensional compressible flow problems. *SIAM Journal on Scientific Computing*, 29(4):1781–1824, 2007a.

Pierre-Henri Maire, Rémi Abgrall, Jérôme Breil, and Jean Ovadia. A Cell-Centered Lagrangian Scheme for Two-Dimensional Compressible Flow Problems. *SIAM Journal on Scientific Computing*, 29(4):1781–1824, January 2007b.

A. Majda. *Compressible Fluid Flow and Systems of Conservation Laws in Several Space Variables*, volume 53 of *Applied Mathematical Sciences*. Springer New York, New York, NY, 1984.

Yu A. Malkov and D. A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, April 2020.

Zhiping Mao, Lu Lu, Olaf Marxen, Tamer A. Zaki, and George Em Karniadakis. Deepm&mnet for hypersonics: Predicting the coupled flow and finite-rate chemistry behind a normal shock using neural-network approximation of operators. *J. Comput. Phys.*, 447:110698, 2021.

Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, June 1963.

A. Marrel, N. Marie, and M. De Lozzo. Advanced surrogate model and sensitivity analysis methods for sodium fast reactor accident assessment. *Reliability Engineering & System Safety*, 138:232–241, 2015.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-student curriculum learning, 2017.

Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

M. D. McKay, R. J. Beckman, and W. J. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21(2):239, May 1979.

D. Mihalas and B. W. Mihalas. *Foundations of radiation hydrodynamics*. Oxford University Press, United States, 1984.

Hossein Mobahi. Training recurrent neural networks by diffusion. *arXiv preprint arXiv:1601.04114*, 2016.

Jonas Mockus. On bayesian methods for seeking the extremum. In *Proceedings of the IFIP Technical Conference*, page 400–404, Berlin, Heidelberg, 1974. Springer-Verlag.

Letif Mones, Noam Bernstein, and Gábor Csányi. Exploration, Sampling, And Reconstruction of Free Energy Surfaces with Gaussian Process Regression. *Journal of Chemical Theory and Computation*, 12(10):5100–5110, October 2016.

Alfred Müller. Integral probability metrics and their generating classes of functions. *Advances in Applied Probability*, 29(2):429–443, 1997.

Habib N. Najm, Peter S. Wyckoff, and Omar M. Knio. A Semi-implicit Numerical Scheme for Reacting Flow. *Journal of Computational Physics*, 143(2):381–402, July 1998.

Y. Nesterov. A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 269:543–547, 1983.

Behnam Neyshabur, Srinadh Bhojanapalli, David Mcallester, and Nati Srebro. Exploring generalization in deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5947–5956. Curran Associates, Inc., 2017.

Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. A PAC-bayesian approach to spectrally-normalized margin bounds for neural networks. In *International Conference on Learning Representations*, 2018.

Jorge Nocedal and Stephen J. Wright, editors. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, 1999.

Frank Noé, Alexandre Tkatchenko, Klaus-Robert Müller, and Cecilia Clementi. Machine learning for molecular simulation. *Annual Review of Physical Chemistry*, 71(1):361–390, April 2020.

Jeremy E. Oakley and Anthony O'Hagan. Probabilistic sensitivity analysis of complex models: a bayesian approach. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 66(3):751–769, 2004.

Bernt Oksendal. *Stochastic Differential Equations (3rd Ed.): An Introduction with Applications*. Springer-Verlag, Berlin, Heidelberg, 1992.

Michael K. Painter, Madhav Erraguntla, Gary L. Hogg, and Brian Beachkofski. Using simulation, data mining, and knowledge discovery techniques for optimized aircraft engine fleet management. In *Proceedings of the 2006 Winter Simulation Conference*, pages 1253–1260, 2006.

Manolis Papadrakakis, Vissarion Papadopoulos, and Nikos D. Lagaros. Structural reliability analyis of elastic-plastic structures using neural networks and monte carlo simulation. *Computer Methods in Applied Mechanics and Engineering*, 136(1):145–163, 1996.

Lorenzo Pareschi. An introduction to uncertainty quantification for kinetic equations and related problems. *arXiv e-prints*, art. arXiv:2004.05072, April 2020.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy,

Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Simon Peluchon. *Approximation numrique et modelisation de l'ablation liquide*. Theses, Université de Bordeaux, November 2017.

Benoit Perthame. *Transport Equations in Biology*. 01 2007.

Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

Nuria Plattner, Stefan Doerr, Gianni De Fabritiis, and Frank Noé. Complete protein–protein association kinetics in atomic detail revealed by molecular dynamics simulations and Markov modelling. *Nature Chemistry*, 9(10):1005–1011, October 2017.

G. Poëtte, B. Després, and D. Lucor. Uncertainty Quantification for Systems of Conservation Laws. *J. Comp. Phys.*, 228(7):2443–2467, 2009.

Gaël Poëtte, David Lugato, and Paul Novello. An analogy between solving Partial Differential Equations with Monte-Carlo schemes and the Optimisation process in Machine Learning (and few illustrations of its benefits). working paper or preprint, April 2021.

B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, January 1964.

Gaël Poëtte. A gPC-intrusive Monte-Carlo scheme for the resolution of the uncertain linear Boltzmann equation. *Journal of Computational Physics*, 385:135–162, May 2019a.

Gaël Poëtte. Contribution to the mathematical andnumerical analysis of uncertain systems of conservation laws and of the linear and nonlinear boltzmann equation. *HDR, Université de Bordeaux*, pages 135–162, 2019b.

Gaël Poëtte. Spectral convergence of the generalized Polynomial Chaos reduced model obtained from the uncertain linear Boltzmann equation. *Mathematics and Computers in Simulation*, 177:24–45, November 2020.

Clémentine Prieur and Stefano Tarantola. *Variance-Based Sensitivity Analysis: Theory and Estimation Algorithms*, pages 1–23. Springer International Publishing, Cham, 2016.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

Haifeng Qian and Mark N. Wegman. L2-nonexpansive neural networks. In *International Conference on Learning Representations*, 2019.

Maxim Raginsky, Alexander Rakhlin, and Matus Telgarsky. Non-convex learning via stochastic gradient langevin dynamics: a nonasymptotic analysis. In Satyen Kale and Ohad Shamir, editors, *Proceedings of the 2017 Conference on Learning Theory*, volume 65 of *Proceedings of Machine Learning Research*, pages 1674–1703. PMLR, 07–10 Jul 2017.

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

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

Saman Razavi, Anthony Jakeman, Andrea Saltelli, Clémentine Prieur, Bertrand Iooss, Emanuele Borgonovo, Elmar Plischke, Samuele Lo Piano, Takuya Iwanaga, William Becker, Stefano Tarantola, Joseph H.A. Guillaume, John Jakeman, Hoshin Gupta, Nicola Melillo, Giovanni Rabitti, Vincent Chabridon, Qingyun Duan, Xifu Sun, Stefán Smith, Razi Sheikholeslami, Nasim Hosseini, Masoud Asadzadeh, Arnald Puy, Sergei Kucherenko, and Holger R. Maier. The future of sensitivity analysis: An essential discipline for systems modeling and policy support. *Environmental Modelling & Software*, 137:104954, 2021.

Markus Reichstein, Gustau Camps-Valls, Bjorn Stevens, Martin Jung, Joachim Denzler, Nuno Carvalhais, and Prabhat. Deep learning and process understanding for data-driven earth system science. *Nature*, 566(7743):195–204, Feb 2019.

Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. *CoRR*, abs/1803.09050, 2018.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

Ribana Roscher, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. Explainable machine learning for scientific insights and discoveries. *IEEE Access*, 8:42200–42216, 2020.

Samuel H. Rudy, Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4), 2017.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. Mcclelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.

Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Andrea Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280 – 297, 2002.

T. J. Santner, Williams B., and Notz W. *The Design and Analysis of Computer Experiments, Second Edition*. Springer-Verlag, 2018.

Elia Schneider, Luke Dai, Robert Q. Topper, Christof Drechsel-Grau, and Mark E. Tuckerman. Stochastic Neural Network Approach for Learning High-Dimensional Free Energy Surfaces. *Physical Review Letters*, 119(15):150601, October 2017.

Nicol N. Schraudolph. Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent. *Neural Computation*, 14(7):1723–1738, July 2002.

James B. Scoggins, Vincent Leroy, Georgios Bellas-Chatzigeorgis, Bruno Dias, and Thierry E. Magin. Mutation++: Multicomponent thermodynamic and transport properties for ionized gases in c++. *SoftwareX*, 12, 2020.

S. Seo, M. Wallat, T. Graepel, and K. Obermayer. Gaussian process regression: Active data selection and test point rejection. In *Proceedings of the International Joint Conference on Neural Networks*, 2000.

Burr Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104:148–175, 2016.

M. C. Shewry and H. P. Wynn. Maximum entropy sampling. *Journal of Applied Statistics*, 14 (2):165–170, January 1987.

Abhinav Shrivastava, Abhinav Gupta, and Ross B. Girshick. Training region-based object detectors with online hard example mining. *CoRR*, abs/1604.03540, 2016.

Jean-François Sigrist. *Numerical Simulation, An Art of Prediction 1*. 12 2019.

Jean-François Sigrist. *Numerical Simulation, An Art of Prediction 2: Examples*. 01 2020.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, page 2951–2959, Red Hook, NY, USA, 2012. Curran Associates Inc.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2171–2180, Lille, France, 07–09 Jul 2015. PMLR.

Ilya M. Sobol. Sensitivity estimates for nonlinear mathematical models. *MMCE*, (1):407–414, 1993.

Le Song, Alex Smola, Arthur Gretton, Karsten M. Borgwardt, and Justin Bedo. Supervised feature selection via dependence estimation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 823–830, New York, NY, USA, 2007. Association for Computing Machinery.

Adrien Spagnol, Rodolphe Le Riche, and Sébastien Da Veiga. Global sensitivity analysis for optimization with variable selection. *SIAM/ASA J. Uncertain. Quantification*, 7:417–443, 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

Thomas Stecher, Noam Bernstein, and Gábor Csányi. Free Energy Surface Reconstruction from Umbrella Samples Using Gaussian Process Regression. *Journal of Chemical Theory and Computation*, 10(9):4079–4097, September 2014.

Curtis B. Storlie and Jon C. Helton. Multiple predictor smoothing methods for sensitivity analysis: Description of techniques. *Reliability Engineering and System Safety*, 93(1):28–54, January 2008.

Curtis B. Storlie, Laura P. Swiler, Jon C. Helton, and Cedric J. Sallaberry. Implementation and evaluation of nonparametric regression procedures for sensitivity analysis of computationally demanding models. *Reliability Engineering & System Safety*, 94:1735–1763, 2009.

G. Strang. On the construction and the comparison of difference schemes. *SIAM, Journal on Numerical Analysis*, 5(3):506–517, 1968.

D.W. Stroock and S.R.S. Varadhan. *Multidimensional Diffusion Processes*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 1997.

Bruno Sudret. Global sensitivity analysis using polynomial chaos expansions. *Reliability Engineering & System Safety*, 93(7):964–979, July 2008.

Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018.

Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017a.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017b.

Laura von Rueden, Sebastian Mayer, Rafet Sifa, Christian Bauckhage, and Jochen Garcke. Combining machine learning and simulation to a hybrid modelling approach: Current and future directions. In Michael R. Berthold, Ad Feelders, and Georg Krempl, editors, *Advances in Intelligent Data Analysis XVIII*, pages 548–560, Cham, 2020. Springer International Publishing.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*, 2019.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *1960 IRE WESCON Convention Record, Part 4*, pages 96–104, New York, 1960. IRE.

N. Wiener. The Homogeneous Chaos. *Amer. J. Math.*, 60:897–936, 1938.

Nick Winovich, Karthik Ramani, and Guang Lin. Convpde-uq: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains. *Journal of Computational Physics*, 394:263 – 279, 2019.

Dongbin Xiu and George Em Karniadakis. The wiener–askey polynomial chaos for stochastic differential equations. *SIAM J. Sci. Comput.*, 24(2):619–644, February 2002.

Da Xu, Yuting Ye, and Chuanwei Ruan. Understanding the role of importance weighting for deep learning. In *International Conference on Learning Representations*, 2021.

Huan Xu and Shie Mannor. Robustness and generalization. *Machine Learning*, 86(3):391–423, Mar 2012.

Lin Yang, Raman Arora, Vladimir braverman, and Tuo Zhao. The physical systems behind optimization algorithms. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Nanyang Ye, Zhanxing Zhu, and Rafal Mantiuk. Langevin dynamics with continuous tempering for training deep neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

Linfeng Zhang, Jiequn Han, Han Wang, Wissam Saidi, Roberto Car, and Weinan E. End-to-end symmetry preserving inter-atomic potential energy model for finite and extended systems. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics*, 394:56 – 81, 2019.

Maxwell I. Zimmerman and Gregory R. Bowman. Fast conformational searches by balancing explorationexploitation trade-offs. *Journal of Chemical Theory and Computation*, 11(12): 5747–5757, December 2015.

Appendices

Appendix A: Demonstrations (Chapter 3)

Illustration of the link using derivatives (Section 3.2.1)

We want to approximate $f : \boldsymbol{x} \to f(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^{n_i}$, $f(\boldsymbol{x}) \in \mathbb{R}^{n_o}$ with a NN $f_{\boldsymbol{\theta}}$. The goal of the approximation problem can be seen as being able to generalize to points not seen during the training. We thus want the generalization error $\mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta})$ to be as small as possible. Given an initial data set $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ drawn from $\mathbf{x} \sim d\mathbb{P}_{\mathbf{x}}$ and $\{f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_N)\}$, and the loss function L being the squared L_2 error, recall that the integrated error $J_{\mathbf{x}}(\boldsymbol{\theta})$, its estimation $\widehat{J_{\mathbf{x}}}(\boldsymbol{\theta})$ and the generalization error $\mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta})$ can be written:

$$\begin{aligned} J_{\mathbf{x}}(\boldsymbol{\theta}) &= \int_{\mathbf{S}} \|f(\boldsymbol{x}) - f_{\boldsymbol{\theta}}(\boldsymbol{x})\| d\mathbb{P}_{\mathbf{x}}, \\ \widehat{J_{\mathbf{x}}}(\boldsymbol{\theta}) &= \frac{1}{N} \sum_{i=1}^{N} \|f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - f(\boldsymbol{x}_i)\|, \\ \mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta}) &= J_{\mathbf{x}}(\boldsymbol{\theta}) - \widehat{J_{\mathbf{x}}}(\boldsymbol{\theta}), \end{aligned} \tag{8.1}$$

where $\|.\|$ denotes the squared L_2 norm. In the following, we find an upper bound for $\mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta})$. We start by finding an upper bound for $J_{\mathbf{x}}(\boldsymbol{\theta})$ and then for $\mathcal{J}_{\mathbf{x}}(\boldsymbol{\theta})$ using equation (8.1).

Let S_i, $i \in \{1, ..., N\}$ be some sub-spaces of a bounded space $\mathbf{S}$ such that $\mathbf{S} = \bigcup_{i=1}^{N} S_i$, $\bigcap_{i=1}^{N} S_i = \emptyset$, and $\boldsymbol{x}_i \in S_i$. Then,

$$J_{\mathbf{x}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \int_{S_i} \|f(\boldsymbol{x}) - f_{\boldsymbol{\theta}}(\boldsymbol{x})\| d\mathbb{P}_{\mathbf{x}},$$
$$J_{\mathbf{x}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \int_{S_i} \|f(\boldsymbol{x}_i + \boldsymbol{x} - \boldsymbol{x}_i) - f_{\boldsymbol{\theta}}(\boldsymbol{x})\| d\mathbb{P}_{\mathbf{x}}.$$

Suppose that $n_i = n_o = 1$ ($\boldsymbol{x}$ becomes x and $\mathbf{x}$ becomes x) and f twice differentiable. Let $|\mathbf{S}| = \int_{\mathbf{S}} d\mathbb{P}_{\mathrm{x}}$. The volume $|\mathbf{S}| = 1$ since $d\mathbb{P}_{\mathrm{x}}$ is a probability measure, and therefore $|S_i| < 1$ for all $i \in \{1, ..., N\}$. Using Taylor expansion at order 2, and since $|S_i| < 1$ for all $i \in \{1, ..., N\}$

$$J_{\mathrm{x}}(\boldsymbol{\theta}) = \sum_{i=1}^{N} \int_{S_i} \|f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 - f_{\boldsymbol{\theta}}(x) + \mathcal{O}((x - x_i)^3)\| d\mathbb{P}_{\mathrm{x}}.$$

To find an upper bound for $J(\boldsymbol{\theta})$, we can first find an upper bound for $|A_i(x)|$, with $A_i(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 - f_{\boldsymbol{\theta}}(x) + \mathcal{O}((x - x_i)^3)$.

NN $f_{\boldsymbol{\theta}}$ is $K_{\boldsymbol{\theta}}-$Lipschitz, so since $\mathbf{S}$ is bounded (so are S_i), for all $x \in S_i$, $|f_{\boldsymbol{\theta}}(x) - f_{\boldsymbol{\theta}}(x_i)| \leq K_{\boldsymbol{\theta}}|x - x_i|$. Hence,

$$\begin{aligned}
&f_{\boldsymbol{\theta}}(x_i) - K_{\boldsymbol{\theta}}|x - x_i| \leq f_{\boldsymbol{\theta}}(x) \leq f_{\boldsymbol{\theta}}(x_i) + K_{\boldsymbol{\theta}}|x - x_i|, \\
&- f_{\boldsymbol{\theta}}(x_i) - K_{\boldsymbol{\theta}}|x - x_i| \leq -f_{\boldsymbol{\theta}}(x) \leq -f_{\boldsymbol{\theta}}(x_i) + K_{\boldsymbol{\theta}}|x - x_i|, \\
&f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i(x - x_i)^2) - f_{\boldsymbol{\theta}}(x_i) - K_{\boldsymbol{\theta}}|x - x_i| + \mathcal{O}((x - x_i)^3) \\
&\leq A_i(x) \leq f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 - f_{\boldsymbol{\theta}}(x_i) + K_{\boldsymbol{\theta}}|x - x_i| + \mathcal{O}((x - x_i)^3), \\
&A_i(x) \leq f(x_i) - f_{\boldsymbol{\theta}}(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 + K_{\boldsymbol{\theta}}|x - x_i| + \mathcal{O}((x - x_i)^3).
\end{aligned}$$

And finally, using triangular inequality,

$$A_i(x) \leq |f(x_i) - f_{\boldsymbol{\theta}}(x_i)| + |f'(x_i)||x - x_i| + \frac{1}{2}|f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}}|x - x_i| + \mathcal{O}(|x - x_i|^3).$$

Now, $\|.\|$ being the squared L_2 norm:

$$
\begin{aligned}
J_{\mathrm{x}}(\boldsymbol{\theta}) &= \sum_{i=1}^{N} \int_{S_i} \| f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2} f''(x_i)(x - x_i)^2 - f_{\boldsymbol{\theta}}(x) + \mathcal{O}(|x - x_i|^3) \| d\mathbb{P}_{\mathrm{x}}, \\
J_{\mathrm{x}}(\boldsymbol{\theta}) &\leq \sum_{i=1}^{N} \int_{S_i} \Bigg[\Big(|f(x_i) - f_{\boldsymbol{\theta}}(x_i)| \Big) + \Big(|f'(x_i)||x - x_i| + \frac{1}{2} |f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}} |x - x_i| \Big) \\
&\qquad + \mathcal{O}(|x - x_i|^3) \Bigg]^2 d\mathbb{P}_{\mathrm{x}}, \\
&= \sum_{i=1}^{N} \int_{S_i} \Bigg[|f(x_i) - f_{\boldsymbol{\theta}}(x_i)|^2 \\
&\qquad + 2|f(x_i) - f_{\boldsymbol{\theta}}(x_i)| \Big(|f'(x_i)||x - x_i| + \frac{1}{2} |f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}} |x - x_i| \Big) \\
&\qquad + \Big[\Big(|f'(x_i)||x - x_i| \Big) + \Big(\frac{1}{2} |f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}} |x - x_i| \Big) \Big]^2 + \mathcal{O}(|x - x_i|^3) \Bigg] d\mathbb{P}_{\mathrm{x}}, \\
&= \sum_{i=1}^{N} \int_{S_i} \Bigg[|f(x_i) - f_{\boldsymbol{\theta}}(x_i)|^2 \\
&\qquad + 2|f(x_i) - f_{\boldsymbol{\theta}}(x_i)| \Big(|f'(x_i)||x - x_i| + \frac{1}{2} |f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}} |x - x_i| \Big) \\
&\qquad + \Big[|f'(x_i)|^2 |x - x_i|^2 + 2K_{\boldsymbol{\theta}} |f'(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}}^2 |x - x_i|^2 \Big] + \mathcal{O}(|x - x_i|^3) \Bigg] d\mathbb{P}_{\mathrm{x}}, \\
&= \sum_{i=1}^{N} \int_{S_i} \Bigg[|f(x_i) - f_{\boldsymbol{\theta}}(x_i)|^2 \\
&\qquad + 2|f(x_i) - f_{\boldsymbol{\theta}}(x_i)| \Big(|f'(x_i)||x - x_i| + \frac{1}{2} |f''(x_i)||x - x_i|^2 + K_{\boldsymbol{\theta}} |x - x_i| \Big) \\
&\qquad + \Big(|f'(x_i)| + K_{\boldsymbol{\theta}} \Big)^2 |x - x_i|^2 + \mathcal{O}(|x - x_i|^3) \Bigg] d\mathbb{P}_{\mathrm{x}}.
\end{aligned}
$$

Hornik's theorem Hornik et al. (1989) states that given a norm $\|.\|_{p,\mu} =$ such that $\|f\|_{p,\mu}^p = \int_{\mathbf{S}} |f(x)|^p d\mu(x)$, with $d\mu$ a probability measure, for any ϵ, there exists $\boldsymbol{\theta}$ such that for a Multi Layer Perceptron, $f_{\boldsymbol{\theta}}$, $\|f(x) - f_{\boldsymbol{\theta}}(x)\|_{p,\mu}^p < \epsilon$,

This theorem grants that for any ϵ, with $d\mu = \sum_{i=1}^{N} \frac{1}{N} \delta(x - x_i)$, there exists $\boldsymbol{\theta}$ such that

$$
\begin{cases}
\|f(x) - f_{\boldsymbol{\theta}}(x)\|_{1,\mu}^1 = \displaystyle\sum_{i=1}^{N} \frac{1}{N} |f(x_i) - f_{\boldsymbol{\theta}}(x_i)| \leq \epsilon, \\
\|f(x) - f_{\boldsymbol{\theta}}(x)\|_{2,\mu}^2 = \displaystyle\sum_{i=1}^{N} \frac{1}{N} \big(f(x_i) - f_{\boldsymbol{\theta}}(x_i) \big)^2 \leq \epsilon.
\end{cases}
\tag{8.2}
$$

Let's introduce i^* such that $i^* = \text{argmin}\,|S_i|$. Note that for any $i \in \{1, ..., N\}$, $\mathcal{O}(|S_i^*|^4)$ is $\mathcal{O}(|S_i|^4)$. Now, let's choose ϵ such that ϵ is $\mathcal{O}(|S_i^*|^4)$. Then, equation (8.2) implies that

$$\begin{cases} |f(x_i) - f_{\boldsymbol{\theta}}(x_i)| = \mathcal{O}(|S_i|^4), \\ \big(f(x_i) - f_{\boldsymbol{\theta}}(x_i)\big)^2 = \mathcal{O}(|S_i|^4), \\ \widehat{J}_{\mathrm{x}}(\boldsymbol{\theta}) = \|f(x) - f_{\boldsymbol{\theta}}(x)\|_{2,\mu}^2 = \mathcal{O}(|S_i|^4). \end{cases}$$

Thus, we have $\mathcal{J}_{\mathrm{x}}(\boldsymbol{\theta}) = J_{\mathrm{x}}(\boldsymbol{\theta}) - \widehat{J}_{\mathrm{x}}(\boldsymbol{\theta}) = J_{\mathrm{x}}(\boldsymbol{\theta}) + \mathcal{O}(|S_i|^4)$ and therefore,

$$\mathcal{J}_{\mathrm{x}}(\boldsymbol{\theta}) \leq \sum_{i=1}^{N} \int_{S_i} \Big[\Big(|f'(x_i)| + K_{\boldsymbol{\theta}} \Big)^2 |x - x_i|^2 d\mathbb{P}_{\mathrm{x}} \Big] + \mathcal{O}(|S_i|^4).$$

Finally,

$$\boxed{\mathcal{J}_{\mathrm{x}}(\boldsymbol{\theta}) \leq \sum_{i=1}^{N} (|f'(x_i)| + K_{\boldsymbol{\theta}})^2 \frac{|S_i|^3}{3} + \mathcal{O}(|S_i|^4).} \tag{8.3}$$

We see that on the regions where $f'(x_i) + K_{\boldsymbol{\theta}}$ is higher, quantity $|S_i|$ (the volume of S_i) has a stronger impact on the GB. Then, since $|S_i|$ can be seen as a metric for the local density of the data set (the smaller $|S_i|$ is, the denser the data set is), the Generalization Bound (GB) can be reduced more efficiently by adding more points around x_i in these regions. This bound also involves $K_{\boldsymbol{\theta}}$, the Lipschitz constant of the NN, which has the same impact as $f'(x_i)$. It also illustrates the link between the Lipschitz constant and the generalization error, which has been pointed out by several works like, for instance, Gouk et al. (2018), Bartlett et al. (2017) and Qian and Wegman (2019).

Problem 1: Unavailability of derivatives (Section 3.3.1)

In this paragraph, we consider $n_i > 1$ but $n_o = 1$. The following derivations can be extended to $n_o > 1$ by applying it to f element-wise. Let $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \epsilon \boldsymbol{I}_{n_i})$ with $\epsilon \in \mathbb{R}^+$ and $\boldsymbol{\epsilon} = (\epsilon_1, ..., \epsilon_{n_i})$, i.e. $\epsilon_i \sim \mathcal{N}(0, \epsilon)$. Using Taylor expansion on f at order 2 gives

$$f(\boldsymbol{x} + \boldsymbol{\epsilon}) = f(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2} \boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3).$$

With $\nabla_x f$ and $\mathbb{H}_x f(\boldsymbol{x})$ the gradient and the Hessian of f w.r.t. $\boldsymbol{x}$. We now compute $Var(f(X + \boldsymbol{\epsilon}))$ and make $Df_\epsilon^2(\boldsymbol{x}) = \epsilon \|\nabla_x f(\boldsymbol{x})\|_F^2 + \frac{1}{2}\epsilon^2 \|\mathbb{H}_{\boldsymbol{x}} f(\boldsymbol{x})\|_F^2$ appear in its expression to establish a link between these two quantities:

$$\begin{aligned} Var(f(\boldsymbol{x} + \boldsymbol{\epsilon})) &= Var\Big(f(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2} \boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3) \Big), \\ &= Var\Big(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2} \boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} \Big) + \mathcal{O}(\|\epsilon\|_2^3). \end{aligned}$$

Since $\epsilon_i \sim \mathcal{N}(0, \epsilon)$, $\boldsymbol{x} = (x_1, ..., x_{n_i})$ and with $\frac{\partial^2 f}{\partial x_i x_j}(\boldsymbol{x})$ the cross derivatives of f w.r.t. x_i and x_j,

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2}\boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} = \sum_{i=1}^{n_i} \epsilon_i \frac{\partial f}{\partial x_i}(\boldsymbol{x}) + \frac{1}{2}\sum_{j=1}^{n_i}\sum_{k=1}^{n_i} \epsilon_j \epsilon_k \frac{\partial^2 f}{\partial x_j x_k}(\boldsymbol{x}),$$

$$\begin{aligned}
Var\Big(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2}\boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon}\Big) =& Var\Big(\sum_{i=1}^{n_i} \epsilon_i \frac{\partial f}{\partial x_i}(\boldsymbol{x}) + \frac{1}{2}\sum_{j=1}^{n_i}\sum_{k=1}^{n_i} \epsilon_j \epsilon_k \frac{\partial^2 f}{\partial x_j x_k}(\boldsymbol{x})\Big), \\
=& \sum_{i_1=1}^{n_i}\sum_{i_2=1}^{n_i} Cov\Big(\epsilon_{i_1}\frac{\partial f}{\partial x_{i_1}}(\boldsymbol{x}), \epsilon_{i_2}\frac{\partial f}{\partial x_{i_2}}(\boldsymbol{x})\Big), \\
&+ \frac{1}{4}\sum_{j_1=1}^{n_i}\sum_{k_1=1}^{n_i}\sum_{j_2=1}^{n_i}\sum_{k_2=1}^{n_i} Cov\Big(\epsilon_{j_1}\epsilon_{k_1}\frac{\partial^2 f}{\partial x_{j_1} x_{k_1}}(\boldsymbol{x}), \epsilon_{j_2}\epsilon_{k_2}\frac{\partial^2 f}{\partial x_{j_2} x_{k_2}}(\boldsymbol{x})\Big) \\
&+ \sum_{i=1}^{n_i}\sum_{j=1}^{n_i}\sum_{k=1}^{n_i} Cov\Big(\epsilon_i\frac{\partial f}{\partial x_i}(\boldsymbol{x}), \epsilon_j\epsilon_k\frac{\partial^2 f}{\partial x_j x_k}(\boldsymbol{x})\Big), \\
=& \sum_{i_1=1}^{n_i}\sum_{i_2=1}^{n_i} \frac{\partial f}{\partial x_{i_1}}(\boldsymbol{x})\frac{\partial f}{\partial x_{i_2}}(\boldsymbol{x}) Cov\Big(\epsilon_{i_1}, \epsilon_{i_2}\Big) \\
&+ \frac{1}{4}\sum_{j_1=1}^{n_i}\sum_{k_1=1}^{n_i}\sum_{j_2=1}^{n_i}\sum_{k_2=1}^{n_i} \frac{\partial^2 f}{\partial x_{j_1} x_{k_1}}(\boldsymbol{x})\frac{\partial^2 f}{\partial x_{j_2} x_{k_2}}(\boldsymbol{x}) Cov\Big(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}\Big) \\
&+ \sum_{i=1}^{n_i}\sum_{j=1}^{n_i}\sum_{k=1}^{n_i} \frac{\partial f}{\partial x_i}(\boldsymbol{x})\frac{\partial^2 f}{\partial x_j x_k}(\boldsymbol{x}) Cov\Big(\epsilon_i, \epsilon_j\epsilon_k\Big).
\end{aligned}$$

In this expression, three quantities have to be assessed : $Cov(\epsilon_{i_1}, \epsilon_{i_2})$, $Cov(\epsilon_i, \epsilon_j\epsilon_k)$ and $Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2})$.

First, since $(\epsilon_1, ..., \epsilon_{n_i})$ are i.i.d.,

$$Cov\Big(\epsilon_{i_1}, \epsilon_{i_2}\Big) = \begin{cases} Var(\epsilon_i) = \epsilon \text{ if } i_1 = i_2 = i, \\ 0 \text{ otherwise.} \end{cases}.$$

To assess $Cov(\epsilon_i, \epsilon_j\epsilon_k)$, three cases have to be considered.

- If $i = j = k$, because $\mathbb{E}[\epsilon_i^3] = 0$,
$$\begin{aligned} Cov(\epsilon_i, \epsilon_j\epsilon_k) &= Cov(\epsilon_i, \epsilon_i^2), \\ &= \mathbb{E}[\epsilon_i^3] - \mathbb{E}[\epsilon_i]\mathbb{E}[\epsilon_i^2], \\ &= 0. \end{aligned}$$

- If $i = j$ or $i = k$ (we consider $i = k$, and the result holds for $i = j$ by commutativity),
$$\begin{aligned} Cov(\epsilon_i, \epsilon_j\epsilon_k) &= Cov(\epsilon_i, \epsilon_i\epsilon_j), \\ &= \mathbb{E}[\epsilon_i^2\epsilon_j] - \mathbb{E}[\epsilon_i]\mathbb{E}[\epsilon_i\epsilon_j], \\ &= \mathbb{E}[\epsilon_i^2]\mathbb{E}[\epsilon_j], \\ &= 0. \end{aligned}$$
- If $i \neq j$ and $i \neq k$, ϵ_i and $\epsilon_j\epsilon_k$ are independent and so $Cov(\epsilon_i, \epsilon_j\epsilon_k) = 0$.

Finally, to assess $Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2})$, four cases have to be considered:

- If $j_1 = j_2 = k_1 = k_2 = i$,
$$\begin{aligned} Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}) &= Var(\epsilon_i^2), \\ &= 2\epsilon^2. \end{aligned}$$
- If $j_1 = k_1 = i$ and $j_2 = k_2 = j$, $Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}) = Cov(\epsilon_i^2, \epsilon_j^2) = 0$ since ϵ_i^2 and ϵ_j^2 are independent.
- If $j_1 = j_2 = j$ and $k_1 = k_2 = k$,
$$\begin{aligned} Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}) &= Var(\epsilon_j\epsilon_k), \\ &= Var(\epsilon_j)Var(\epsilon_k), \\ &= \epsilon^2. \end{aligned}$$
- If $j_1 \neq k_1, j_2$ and k_2,
$$\begin{aligned} Cov(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}) &= \mathbb{E}[\epsilon_{j_1}\epsilon_{k_1}\epsilon_{j_2}\epsilon_{k_2}] - \mathbb{E}[\epsilon_{j_1}\epsilon_{k_1}]\mathbb{E}[\epsilon_{j_2}\epsilon_{k_2}], \\ &= \mathbb{E}[\epsilon_{j_1}]\mathbb{E}[\epsilon_{k_1}\epsilon_{j_2}\epsilon_{k_2}] - \mathbb{E}[\epsilon_{j_1}]\mathbb{E}[\epsilon_{k_1}]\mathbb{E}[\epsilon_{j_2}\epsilon_{k_2}], \\ &= 0. \end{aligned}$$

All other possible cases can be assessed using the previous results, commutativity and symmetry of Cov operator. Hence,
$$\begin{aligned} Var\Big(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon} + \frac{1}{2}\boldsymbol{\epsilon}^T \cdot \mathbb{H}_x f(\boldsymbol{x}) \cdot \boldsymbol{\epsilon}\Big) =& \sum_{i_1=1}^{n_i}\sum_{i_2=1}^{n_i} \frac{\partial f}{\partial x_{i_1}}(\boldsymbol{x})\frac{\partial f}{\partial x_{i_2}}(\boldsymbol{x}) Cov\Big(\epsilon_{i_1}, \epsilon_{i_2}\Big) \\ &+ \frac{1}{4}\sum_{j_1=1}^{n_i}\sum_{k_1=1}^{n_i}\sum_{j_2=1}^{n_i}\sum_{k_2=1}^{n_i} \frac{\partial^2 f}{\partial x_{j_1}x_{k_1}}(\boldsymbol{x})\frac{\partial^2 f}{\partial x_{j_2}x_{k_2}}(\boldsymbol{x}) Cov\Big(\epsilon_{j_1}\epsilon_{k_1}, \epsilon_{j_2}\epsilon_{k_2}\Big), \\ =& \sum_{i=1}^{n_i} \epsilon\frac{\partial f^2}{\partial x_i}(\boldsymbol{x}) + \frac{1}{2}\sum_{j=1}^{n_i}\sum_{k=1}^{n_i} \epsilon^2\frac{\partial^2 f^2}{\partial x_j x_k}(\boldsymbol{x}), \\ =& \epsilon\|\nabla_x f(\boldsymbol{x})\|_F^2 + \frac{1}{2}\epsilon^2\|\mathbb{H}_{\boldsymbol{x}} f(\boldsymbol{x})\|_F^2, \\ =& Df_\epsilon^2(\boldsymbol{x}). \end{aligned}$$

And finally,

$$\boxed{Var(f(\boldsymbol{x}+\boldsymbol{\epsilon})) = Df_{\epsilon}^{2}(\boldsymbol{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3)} \tag{8.4}$$

If we consider $\widehat{Df^2}_{\epsilon}(\boldsymbol{x})$ as defined in equation (3.2), on section* 3.2.2 of the main document, $\widehat{Df^2}_{\epsilon}(\boldsymbol{x}) \underset{k\to\infty}{\to} Var(f(\boldsymbol{x}+\boldsymbol{\epsilon}))$. Since $Var(f(\boldsymbol{x}+\boldsymbol{\epsilon})) = Df_{\boldsymbol{\epsilon}}^{2}(\boldsymbol{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3)$, $\widehat{Df^2}_{\epsilon}(\boldsymbol{x})$ is a biased estimator of $Df_{\boldsymbol{\epsilon}}^{2}(\boldsymbol{x})$, with bias $\mathcal{O}(\|\boldsymbol{\epsilon}\|_2^3)$. Hence, when $\epsilon \to 0$, $\widehat{Df^2}_{\epsilon}(\boldsymbol{x})$ becomes an unbiased estimator of $Df_{\boldsymbol{\epsilon}}^{2}(\boldsymbol{x})$.

Appendix B: Hyperparameter spaces

Hyperparameters of Chapter 3

The values chosen for the hyperparameters of Chapter 3 experiments are gathered in Table 8.1. For Adam optimizer hyperparameters, we kept the default values of Keras implementation. We chose these hyperparameters after simple grid searches.

Experiment	m	k	learning rate	batch size	epochs	optimizer	random seeds
double moon	100	20	1×10^{-3}	100	10000	SGD	50
Boston housing	8	35	5×10^{-4}	404	50000	Adam	10
Breast Cancer	50	35	5×10^{-2}	455	250000	Adam	10
MNIST	40	20	1×10^{-3}	25	25	Adam	40
Cifar10	40	20	1×10^{-3}	25	25	Adam	50
RTE	20	10	3×10^{-4}	8	10000	Adam	50
STS-B	30	30	3×10^{-4}	8	10000	Adam	50
MRPC	75	25	3×10^{-4}	16	10000	Adam	50

Table 8.1: Hyperparameters values for experiments of Chapter 3

Hyperparameters of Chapter 4

In this section, we describe hyperparameters spaces used for each problem in this chapter. Note that hyperparameter `n_seeds` denotes the number of random repetitions of the training for each hyperparameter configuration. If a conditional hyperparameter X_j is only involved for some specific values of a main hyperparameter X_i, it is displayed with an indent on tab lines below that of X_i, with the value of X_i required for X_j to be involved in the training.

Runge and MNIST

For Runge and MNIST, only fully connected Neural Networks are trained, and the width (`n_units`) is the same for every layer.

Conditional groups: (see **(iii)** of Section 4.3.3) $\mathcal{G}_0$ and $\mathcal{G}_{\texttt{dropout\_rate}}$

hyperparameter	type	values for Runge	values for MNIST
`n_layers`	integer	$\in \{1, ..., 10\}$	same
`n_units`	integer	$\in \{7, ..., 512\}$	$\in \{128, ..., 1500\}$
`activation`	categorical	`elu`, `relu`, `tanh` or `sigmoid`	same
`dropout`	boolean	`true` or `false`	same
`yes:dropout_rate`	continuous	$\in [0, 1]$	same
`batch_norm`	boolean	`true` or `false`	same
`weights_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$	same
`weights_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$	same
`bias_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$	same
`bias_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$	same
`batch_size`	integer	$\in \{1, ..., 11\}$	$\in \{1, ..., 256\}$
`loss_function`	categorical	L_2 error or L_1 error	L_2 error or crossentropy
`optimizer`	categorical	`adam`, `sgd`, `rmsprop` or `adagrad`	same
`n_seeds`	integer	$\in \{1, ..., 40\}$	$\in \{1, ..., 10\}$

Table 8.2: Hyperparameters values for Runge & MNIST

Bateman

For Bateman, only fully connected Neural Networks are trained, and the width (`n_units`) is the same for every layer.

hyperparameter	type	values for Bateman
`n_layers`	integer	$\in \{1, ..., 10\}$
`n_units`	integer	$\in \{7, ..., 512\}$
`activation`	categorical	`elu`, `relu`, `tanh` or `sigmoid`
`dropout`	boolean	`true` or `false`
`yes:dropout_rate`	continuous	$\in [0, 1]$
`batch_norm`	boolean	`true` or `false`
`learning_rate`	continuous	$\in [1 \times 10^{-6}, 1 \times 10^{-2}]$
`weights_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`weights_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`batch_size`	integer	$\in \{1, ..., 500\}$
`loss_function`	categorical	L_2 error or L_1 error
`optimizer`	categorical	`adam`, `sgd`, `rmsprop`, `adagrad` or `nadam`
`adam:amsgrad`	boolean	`true` or `false`
`adam, nadam:1st_moment_decay`	continuous	$\in [0.8, 1]$
`adam, nadam:2nd_moment_decay`	continuous	$\in [0.8, 1]$
`rmsprop:centered`	boolean	`true` or `false`
`sgd:nesterov`	boolean	`true` or `false`
`sgd, rmsprop:momentum`	continuous	$\in [0.5, 0.99]$
`n_seeds`	integer	$\in \{1, ..., 10\}$

Table 8.3: Hyperparameters values for Bateman

Conditional groups: (see **(iii)** of Section 4.3.3) $\mathcal{G}_0$, $\mathcal{G}_{\texttt{dropout\_rate}}$, $\mathcal{G}_{\texttt{amsgrad}}$, $\mathcal{G}_{\texttt{centered}}$, $\mathcal{G}_{\texttt{nesterov}}$, $\mathcal{G}_{\texttt{momentum}}$ and
$\mathcal{G}_{(\texttt{1st\_moment},\texttt{2nd\_moment})}$

Cifar10

For Cifar10, we use Convolutional Neural Networks, whose width increases with the depth according to hyperparameters `stages` and `stage_mult`. The first layer has width `n_filters`, and then, `stages` -1 times, the network is widen by a factor `stage_mult`. For instance, a neural network with `n_filters` $= 20$, `n_layers` $= 3$, `stages` $= 3$ and `stage_mult` $= 2$ will have a first layer with 20 filters, a second layer with `n_filters` $\times$ *stage_mult* $= 40$ filters, and a third layer with `n_filters` $\times$ `stage_mult`$^{\texttt{stages}-1} = 60$ filters.

hyperparameter	type	values for Cifar10
`n_layers`	integer	$\in \{3, ..., 12\}$
`n_filters`	integer	$\in \{16, ..., 100\}$
`stages`	integer	$\in \{1, 4\}$
`stage_mult`	continuous	$\in [1, 3]$
`kernel_size`	integer	$\in \{1, 5\}$
`pool_size`	integer	$\in \{2, 5\}$
`pool_type`	categorical	`max` or `average`
`activation`	categorical	`elu`, `relu`, `tanh` or `sigmoid`
`dropout`	boolean	`true` or `false`
` yes:dropout_rate`	continuous	$\in [0, 1]$
`batch_norm`	boolean	`true` or `false`
`learning_rate`	continuous	$\in [1 \times 10^{-6}, 1 \times 10^{-2}]$
`weights_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`weights_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`batch_size`	integer	$\in \{10, ..., 128\}$
`loss_function`	categorical	L_2 error or crossentropy
`optimizer`	categorical	`adam`, `sgd`, `rmsprop`, `adagrad` or `nadam`
` adam:amsgrad`	boolean	`true` or `false`
` adam, nadam:1st_moment_decay`	continuous	$\in [0.8, 1]$
` adam, nadam:2nd_moment_decay`	continuous	$\in [0.8, 1]$
` rmsprop:centered`	boolean	`true` or `false`
` sgd:nesterov`	boolean	`true` or `false`
` sgd, rmsprop:momentum`	continuous	$\in [0.5, 0.99]$
`n_seeds`	integer	$\in \{1, ..., 10\}$

Table 8.4: Hyperparameters values for Cifar10

Conditional groups: (see **(iii)** of Section 4.3.3) $\mathcal{G}_0$, $\mathcal{G}_{\texttt{dropout\_rate}}$, $\mathcal{G}_{\texttt{amsgrad}}$, $\mathcal{G}_{\texttt{centered}}$, $\mathcal{G}_{\texttt{nesterov}}$, $\mathcal{G}_{\texttt{momentum}}$ and $\mathcal{G}_{(\texttt{1st\_moment},\texttt{2nd\_moment})}$

Hyperparameters of Chapter 6

for approximating Mutation++, only fully connected Neural Networks are trained, and the width (`n_units`) is the same for every layer. The conditional groups are the same as Bateman.

hyperparameter	type	values
`n_layers`	integer	$\in \{1, ..., 10\}$
`n_units`	integer	$\in \{7, ..., 512\}$
`activation`	categorical	`elu`, `relu`, `tanh` or `sigmoid`
`dropout`	boolean	`true` or `false`
`yes:dropout_rate`	continuous	$\in [0, 1]$
`batch_norm`	boolean	`true` or `false`
`learning_rate`	continuous	$\in [1 \times 10^{-6}, 1 \times 10^{-2}]$
`weights_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`weights_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l1`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`bias_reg_l2`	continuous	$\in [1 \times 10^{-6}, 0.1]$
`batch_size`	integer	$\in \{1, ..., 500\}$
`loss_function`	categorical	L_2 error or L_1 error
`optimizer`	categorical	`adam`, `sgd`, `rmsprop`, `adagrad` or `nadam`
`adam:amsgrad`	boolean	`true` or `false`
`adam, nadam:1st_moment_decay`	continuous	$\in [0.8, 1]$
`adam, nadam:2nd_moment_decay`	continuous	$\in [0.8, 1]$
`rmsprop:centered`	boolean	`true` or `false`
`sgd:nesterov`	boolean	`true` or `false`
`sgd, rmsprop:momentum`	continuous	$\in [0.5, 0.99]$
`n_seeds`	integer	$\in \{1, ..., 10\}$

Table 8.5: Hyperparameters values for Mutation++ approximation

Appendix C: Construction of Bateman data set (Chapter 4)

Bateman data set is based on the resolution of the Bateman equations, which is an ODE system modeling multi species reactions:

$$\partial_t \boldsymbol{\eta}(t) = \boldsymbol{\Sigma_r}\big(\boldsymbol{\eta}(t)\big) \cdot \boldsymbol{\eta}(t), \text{ with initial conditions } \boldsymbol{\eta}(0) = \boldsymbol{\eta_0},$$

and $\boldsymbol{\eta} \in (\mathbb{R}^+)^M$, $\boldsymbol{\Sigma}_r \in \mathbb{R}^{M\times M}$. Here, $f : (\boldsymbol{\eta_0}, t) \to \boldsymbol{\eta}(t)$, and we are interested in $\boldsymbol{\eta}(t)$, which is the concentration of each of the species S_k, with $k \in \{1, ..., M\}$. For physical applications, M ranges from tens to thousands. We consider the particular case $M = 11$. Matrix $\boldsymbol{\Sigma_r}\big(\boldsymbol{\eta}(t)\big)$ depends on reaction constants. Here, 4 reactions are considered and each reaction p has constant σ_p.

$$\begin{cases} (1) : S_1 + S_2 \to S_3 + S_4 + S_6 + S_7, \\ (2) : S_3 + S_4 \to S_2 + S_8 + S_{11}, \\ (3) : S_2 + S_{11} \to S_3 + S_5 + S_9, \\ (4) : S_3 + S_{11} \to S_2 + S_5 + S_6 + S_{10}, \end{cases}$$

with $\sigma_1 = 1$, $\sigma_2 = 5$, $\sigma_3 = 3$ and $\sigma_4 = 0.1$. To obtain $\boldsymbol{\Sigma_r}\big(\boldsymbol{\eta}(t)\big)$, the species have to be considered one by one. Here we give an example of how to construct the second row of $\boldsymbol{\Sigma_r}\big(\boldsymbol{\eta}(t)\big)$. The other rows are built the same way. Given the reaction equations :

$$\partial_t \eta_2 = -\sigma_1\eta_1\eta_2 + \sigma_2\eta_3\eta_4 - \sigma_3\eta_2\eta_{11} + \sigma_4\eta_3\eta_{11},$$

because S_2 disappears in reactions (1) and (3) involving S_1 and S_{11} as other reactants at rate σ_1 and σ_3, respectively, and appears in reactions (2) and (4) involving S_3, S_4 and S_3, S_{11} as reactants, at rate σ_2 and σ_4 respectively. Hence, the second row of $\boldsymbol{\Sigma_r}\big(\boldsymbol{\eta}(t)\big)$ is

$$[0,\ -\sigma_1\eta_1,\ 0,\ \sigma_2\eta_3,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ -\sigma_3\eta_2 + \sigma_4\eta_3],$$

with $\boldsymbol{\eta}(t)$ denoted by $\boldsymbol{\eta}$ to simplify the equation and η_i the i-th component of $\boldsymbol{\eta}$.

To construct the training, validation and test data sets, we sample uniformly $(\boldsymbol{\eta}_0, t) \in [0, 1]^{12} \times [0, 5]$ 130000 times. We denote these samples $(\boldsymbol{\eta}_0, t)_i$ for $i \in \{1, ..., 130000\}$. Then, we apply a first order Euler solver with a time step of 10^{-3} to compute $f((\boldsymbol{\eta}_0, t)_i)$. As a result, neural network's input is $(\boldsymbol{\eta}_0, t)$ and neural network's output is $f((\boldsymbol{\eta}_0, t))$.

Appendix D: HSICs for conditional hyperparameters (Chapter 4)

MNIST

For MNIST, there is only one conditional hyperparameter, `dropout_rate`, so only one conditional group to consider in order to assess the importance of conditional hyperparameters.

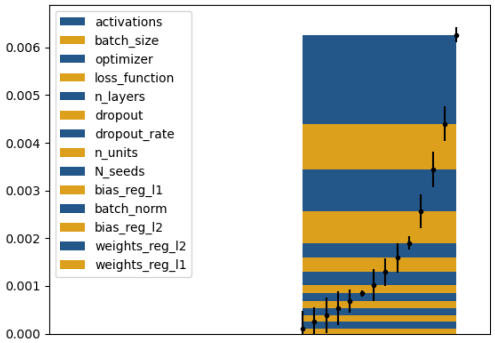
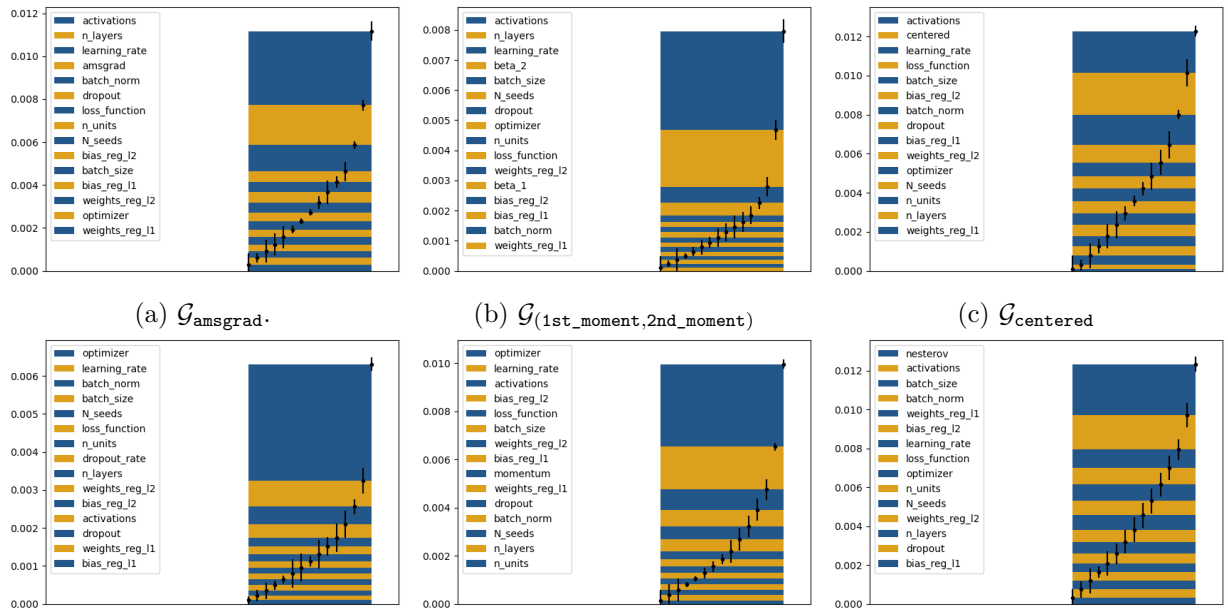


Figure 8.1: HSICs for $\mathcal{G}_{\texttt{dropout\_rate}}$ of MNIST hyperparameter analysis. Conditional hyperparameter `dropout_rate` is not impactful.

Bateman

For Bateman, there are seven conditional hyperparameter, `amsgrad`, `1st_moment` (`beta_1`), `2nd_moment` (`beta_2`), `dropout_rate`, `centered`, `momentum`, and `nesterov`. Six conditional groups, specified in Figure 8.2, have to be considered in order to assess their importance.



(a) $\mathcal{G}_{\texttt{amsgrad}}$.

(b) $\mathcal{G}_{(\texttt{1st\_moment},\texttt{2nd\_moment})}$

(c) $\mathcal{G}_{\texttt{centered}}$

(d) $\mathcal{G}_{\texttt{dropout\_rate}}$

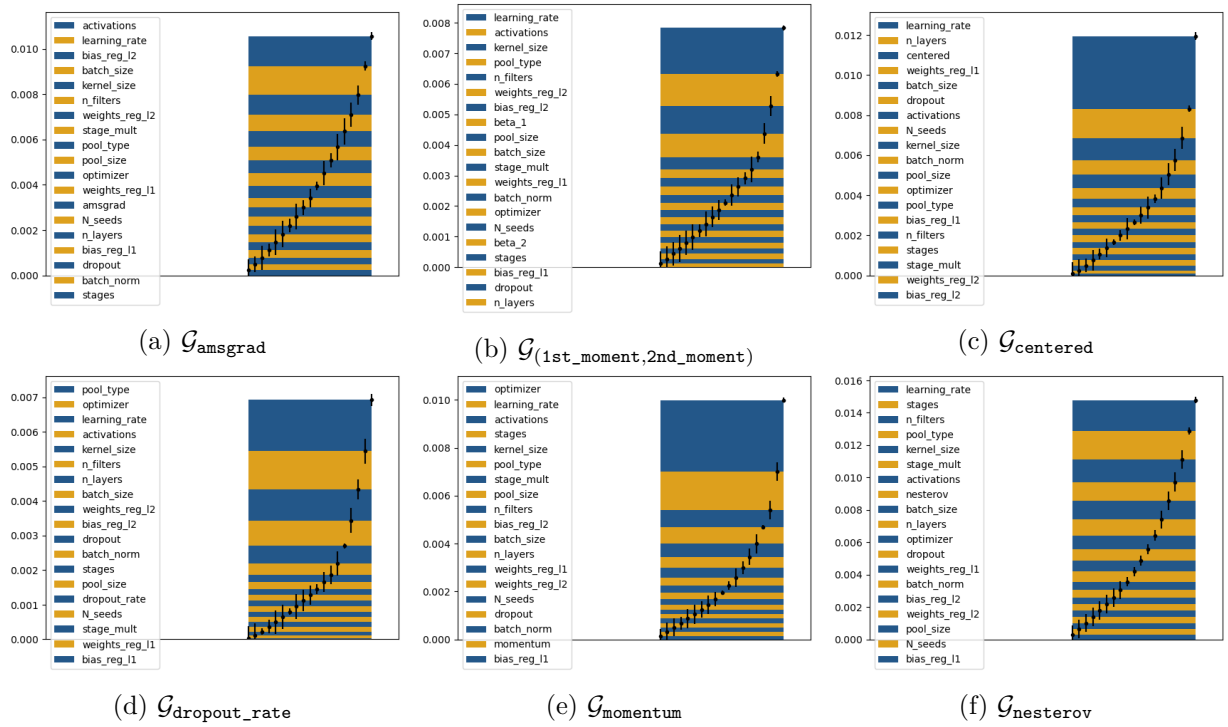
(e) $\mathcal{G}_{\texttt{momentum}}$

(f) $\mathcal{G}_{\texttt{nesterov}}$

Figure 8.2: HSICs for conditional groups of Bateman hyperparameter analysis. (a): `amsgrad` is not impactful (it is in the estimation noise), (b): `1st_moment` is not impactful but `2nd_moment` is the fourth most impactful hyperparameter of this group, (c): `centered` is the second most impactful hyperparameter of this group, (d): `dropout_rate` is not impactful, (e): `momentum` is not impactful, (f): `nesterov` is the most impactful hyperparameter of this group.

Cifar10

For Cifar10, there are seven conditional hyperparameter, `amsgrad`, `1st_moment` (`beta_1`), `2nd_moment` (`beta_2`), `dropout_rate`, `centered`, `momentum`, and `nesterov`. Six conditional groups, specified in Figure 8.3, have to be considered in order to assess their importance.



(a) $\mathcal{G}_{\texttt{amsgrad}}$ (b) $\mathcal{G}_{(\texttt{1st\_moment},\texttt{2nd\_moment})}$ (c) $\mathcal{G}_{\texttt{centered}}$

(d) $\mathcal{G}_{\texttt{dropout\_rate}}$ (e) $\mathcal{G}_{\texttt{momentum}}$ (f) $\mathcal{G}_{\texttt{nesterov}}$

Figure 8.3: HSICs for conditional groups of cifar10 hyperparameter analysis. (a): `amsgrad` is not impactful, (b): `1st_moment`, `2nd_moment` are not impactful, (c): `centered` is the third most impactful hyperparameter of this group, (d): `dropout_rate` is not impactful, (e): `momentum` is not impactful, (f): `nesterov` is not impactful.

ECOLE
DOCTORALE
DE MATHEMATIQUES
HADAMARD

Titre : Contributions à la combinaison entre apprentissage profond supervisé et calcul scientifique, application à la simulation de dynamique des fluides.

Mots clés : Apprentissage automatique scientifique, apprentissage profond, calcul scientifique, analyse statistique

Résumé : Dans cette thèse, nous nous interessons a l'utilisation de l'apprentissage profond pour accelerer des simulations numeriques. Pour atteindre cet objectif, nous nous concentrons sur l'approximation de certaines parties des logiciels de simulation bases sur des Equations Differentielles Partielles (EDP) par un reseau de neurones. La méthodologie proposée s'appuie sur la construction d'un ensemble de donnees, la sélection et l'entrainement d'un reseau de neurones et son integration dans le logiciel original, donnant lieu a une simulation numerique hybride. Malgré la simplicité apparente de cette approche, le contexte des simulations numeriques implique des difficultes specifiques liées à un compromis ominiprésent entre précisions et performances. Afin de satisfaire ces enjeux, nous etudions en detail chaque etape de la methodologie d'apprentissage profond. Ce faisant, nous mettons en evidence certaines similitudes entre l'apprentissage automatique et la simulation numerique, nous permettant de presenter des contributions ayant un impact sur chacun de ces domaines.
Nous identifions les principales etapes de la methodologie d'apprentissage profond comme etant la constitution d'un ensemble de donnees d'entrainement, le choix des hyperparametres d'un reseau de neurones et son entrainement. Pour la premiere etape, nous tirons parti de la possibilite d'echantillonner les donnees d'entrainement à l'aide du logiciel de simulation initial pour caracteriser une distribution d'entrainement plus efficace basee sur la variation locale de la fonction à approcher. Nous generalisons cette observation pour permettre son application a des problemes variés d'apprentissage automatique en construisant une methodologie de ponderation des donnees appelee "Variance Based Sample Weighting". Dans un deuxieme temps, nous proposons l'usage de l'analyse de sensibilite, une approche largement utilisee en calcul scientifique, pour l'optimisation des hyperparametres des reseaux de neurones. Cette approche repose sur l'evaluation qualitative de l'effet des hyperparametres sur les performances d'un reseau de neurones à l'aide du critère d'indépendance de Hilbert-Schmidt. Son adaptation au contexte de l'optimisation des hyperparametres conduit a une methodologie interpretable permettant de construire des reseaux de neurones a la fois performants et precis. Pour la troisieme etape, nous definissons formellement une analogie entre la resolution stochastique d'EDPs et le processus d'optimisation en jeu lors de l'entrainement d'un reseau de neurones. Cette analogie permet d'obtenir un cadre pour l'entrainement des reseaux de neurones base sur la theorie des EDPs, qui ouvre de nombreuses possibilites d'ameliorations pour les algorithmes d'optimisation existants. Enfin, nous appliquons ces methodologies a une simulation numerique de dynamique des fluides couplée à des réactions chimiques multi-espèces.

Title : Combining supervised deep learning and scientific computing: some contributions and application to computational fluid dynamics.

Keywords : Scientific machine learning, deep learning, scientific computing, statistical analysis

Abstract : This thesis settles in the high-stakes emerging field of Scientific Machine Learning which studies the application of machine learning to scientific computing. More specifically, we consider the use of deep learning to accelerate numerical simulations. We focus on approximating some components of Partial Differential Equation (PDE) based simulation softwares by a neural network. This idea boils down to constructing a data set, selecting and training a neural network, and embedding it into the original code, resulting in a hybrid numerical simulation. Although this approach may seem trivial at first glance, the context of numerical simulations comes with several challenges stemming from an accuracy-performances trade-off. To tackle these challenges, we thoroughly study each step of the deep learning methodology while considering the aforementioned constraints. By doing so, we emphasize interplays between numerical simulations and machine learning that can benefit each of these fields.
We identify the main steps of the deep learning methodology as the construction of the training data set, the choice of the hyperparameters of the neural network, and its training. For the first step, we leverage the ability to sample training data with the original software to characterize a more efficient training distribution based on the local variation of the function to approximate. We generalize this approach to general machine learning problems by deriving a data weighting methodology called Variance Based Sample Weighting. For the second step, we introduce the use of sensitivity analysis, an approach widely used in scientific computing, to tackle neural network hyperparameter optimization. This approach is based on qualitatively assessing the effect of hyperparameters on the performances of a neural network using Hilbert-Schmidt Independence Criterion. We adapt it to the hyperparameter optimization context and build an interpretable methodology that yields competitive and cost-effective networks. For the third step, we formally define an analogy between the stochastic resolution of PDEs and the optimization process at play when training a neural network. This analogy leads to a PDE-based framework for training neural networks that opens up many possibilities for improving existing optimization algorithms. Finally, we apply these contributions to a computational fluid dynamics simulation coupled with a multi-species chemical equilibrium library.

Institut Polytechnique de Paris
91120 Palaiseau, France