ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seminar for
Applied
Mathematics

SAM

A Multi-level procedure for enhancing accuracy of machine learning algorithms

K. Lye and S. Mishra and R. Molinaro

A Multi-level procedure for enhancing accuracy of machine learning algorithms.

Kjetil O. Lye [*], Siddhartha Mishra [†] and Roberto Molinaro [‡]

September 19, 2019

Abstract

We propose a multi-level method to increase the accuracy of machine learning algorithms for approximating observables in scientific computing, particularly those that arise in systems modeled by differential equations. The algorithm relies on judiciously combining a large number of computationally cheap training data on coarse resolutions with a few expensive training samples on fine grid resolutions. Theoretical arguments for lowering the generalization error, based on reducing the variance of the underlying maps, are provided and numerical evidence, indicating significant gains over underlying single-level machine learning algorithms, are presented. Moreover, we also apply the multi-level algorithm in the context of forward uncertainty quantification and observe a considerable speed-up over competing algorithms.

1 Introduction

A fundamental goal in scientific computing is the efficient simulation of *observables*, also referred to as functionals, quantities of interest or figures of merit, of systems that arise in physics and engineering. Often, the underlying system is modelled by nonlinear partial differential equations (PDEs). A prototypical example is provided by the simulation of flows past aerospace vehicles where the observables of interest are body forces such as the lift and the drag coefficients and the underlying PDEs are the compressible Euler or Navier-Stokes equations of fluid dynamics. Other interesting examples include the run-up height for a tsunami (with the shallow water equations modeling the flow) or loads (stresses) on structures, with the underlying system being modeled by the equations of elasticity or visco-elasticity.

Computing observables involves first solving the underlying PDE by suitable numerical methods such as finite difference, finite element, finite volume or spectral methods and then evaluating the corresponding observable by another numerical method, usually a quadrature rule.

The computation of observables can be very expensive as solving the underlying (nonlinear) PDE, especially in three space dimensions, entails a large computational cost, even on state of the art high performance computing (HPC) platforms.

This high computational cost is particularly evident when one considers large scale problems such as uncertainty quantification, (Bayesian) inverse problems, data assimilation or optimal control/design. All these problems are of the *many query* type i.e, the underlying PDE has to be solved for a very large number of instances, each corresponding to a particular realization of the input parameter space, in order to compute the *input parameters to observable* map. Querying the computationally costly PDE solver multiple times renders these problems prohibitively expensive.

Although many methods such as reduced order models [35] have been developed to provide a *surrogate* for the PDE solver in computing observables, they may not be stable or accurate enough for complex

---

[*]Seminar for Applied Mathematics (SAM), D-Math
ETH Zürich, Rämistrasse 101, Zürich-8092, Switzerland
[†]Seminar for Applied Mathematics (SAM), D-Math
ETH Zürich, Rämistrasse 101, Zürich-8092, Switzerland
[‡]D-Mavt,ETH Zürich, Rämistrasse 101, Zürich-8092, Switzerland .

non-linear PDEs such as those modeling fluid flows. Hence, there is a pressing need for the design of fast and accurate surrogates for computing observables.

Machine learning is a very popular field within computer science in recent years. In particular, artificial neural networks i.e, layers of units (neurons) that compose affine transformations with simple (scalar) nonlinearities, are a very effective tool in a variety of contexts. *Deep learning*, based on artificial neural networks with a large number of hidden layers, is extremely successful at diverse tasks, for instance in image processing, computer vision, text and speech recognition, game intelligence and more recently in protein folding [7], see [17] and references therein for more applications of deep learning. A key element in *supervised* deep learning is the *training* of tunable parameters in the underlying neural network by (approximately) minimizing suitable *loss functions* on the set of training data. The resulting very high-dimensional (non-convex) optimization problem is customarily solved with variants of the stochastic gradient descent method [32].

Machine learning is being increasingly used in the context of scientific computing. Given that neural networks are very powerful universal function approximators [3, 36], it is natural to consider the space of neural networks as an ansatz space for approximating solutions of PDEs. First proposed in [16] on an underlying collocation approach, it has been successfully used recently in different contexts. See [27, 28, 24, 4, 5, 11] and references therein. This approach appears to work quite well for problems with high regularity (smoothness) of the underlying solutions and/or if the solution of the underlying PDE possesses a representation formula in terms of integrals.

Another set of methods embed deep learning modules within existing numerical methods to improve them. Examples include solving the elliptic equations in a divergence projection step in incompressible flows [34] or learning troubled cell indicators within an RKDG code for applying limiters [30] or recasting finite difference (volume) schemes as neural networks and training the underlying parameters to improve accuracy on coarse grids [21].

In a recent paper [19], the authors developed a deep learning algorithm to approximate observables in computational fluid dynamics. A crucial part of the paper was the identification of challenges in applying deep learning algorithms in the context of PDEs. One of these challenges stemmed from estimates on the so-called *generalization error* (see 2.26 of [19] or (2.15) for the definition), that measures the accuracy of the trained network on unseen inputs. An upper bound on the generalization error in the context of regression of functions, for randomly selected training data is provided by

$$\mathcal{E}_G \sim \mathcal{E}_T + \frac{U}{\sqrt{N}}, \tag{1.1}$$

with $\mathcal{E}_G$ being the generalization error, $\mathcal{E}_T$ the training error (see (2.14) for definition) and N being the number of training samples. The detailed estimate, presented in section 2, (2.19), bounds U in (1.1) in terms of two components, one arising from the trained neural network and the other being proportional to the *standard deviation* of the underlying function. The bound (1.1) clearly illustrates the challenges of using deep learning in the context of approximating observables of PDEs. As long as the upper bound $U \sim \mathcal{O}(1)$, we need a large number of training samples in order to obtain reasonably small generalization errors. Since the training samples are generated by solving PDEs, generation of a large number of training samples necessitates a very high computational cost.

In [19], the authors proposed reducing the generalization error in this *data poor* regime, by selecting training points based on *low-discrepancy sequences*, such as Halton or Sobol sequences which are heavily used in the context of Quasi-Monte Carlo (QMC) methods [2]. This approach yielded considerable increase in accuracy at the same cost, over choosing random training points and allowed the authors to obtain very low prediction errors with a few ($\mathcal{O}(100)$)) training samples for problems such as predicting drag and lift for flows past airfoils.

However, one can only prove that such an approach of using low-discrepancy sequences reduces the generalization error if the underlying function is sufficiently regular. In general, observables arising in computational fluid dynamics for instance, have rather low regularity (see section 2.3.2 of [19]). Moreover, the approach of using low-discrepancy sequences is viable only if the dimension of the underlying input parameter space is only moderately high.

In this paper, we propose another approach for increasing the accuracy (by reducing the generalization error) of deep neural networks for computing observables. Our approach is based on the observation that

the upper bound U in (1.1) involves the standard deviation of the underlying observable (see (2.19) for an exact statement of the estimate). We will reduce the variance (standard deviation) of the observable in order to lower the generalization error. To this end, we adapt a *multi-level* or multi-resolution procedure to the context of machine learning.

Multi-level methods were introduced in the context of numerical quadrature by Heinrich in [13] and for numerical solutions of stochastic differential equations (SDEs) by Giles in [8]. They have been heavily used in recent years for uncertainty quantification in PDEs, solutions of stochastic PDEs, data assimilation and Bayesian inversion. See [9] for a detailed survey of multi-level Monte Carlo methods and applications and [22, 23] for applications to computational fluid dynamics. We would like to point out that multi-level methods are inspired by multi-grid and multi-resolution techniques, which have been used in numerical analysis over many decades.

The basic idea of our multi-level machine learning algorithm is to approximate the observable on a sequence of nested mesh resolutions for solving the underlying PDE. We then learn the so-called *details* (differences of the observable on successive mesh resolutions), instead of the observable itself. If the underlying numerical method converges to a solution of the PDE, then the standard deviation of the details is significantly smaller than the standard deviation of the underlying observable, resulting in a smaller value of the upper bound in (1.1) and allowing us to learn the details with significantly fewer training samples. By carefully balancing the standard deviation of the details with the computational cost of generating the observable at each level of resolution, we aim to reduce the overall generalization error, while keeping the cost of generating the training samples small.

The main aim of this paper is to present this novel multi-level machine learning algorithm for computing observables and to demonstrate the gain in accuracy over standard supervised deep learning algorithms, such as the one proposed in [19] . We will also use this algorithm in the context of speeding up forward uncertainty quantification. Numerical experiments for two prototypical problems in scientific computing will be presented in order to illustrate the gain in efficiency with the proposed algorithm.

The rest of the paper is organized as follows: in section 2, we present the deep learning algorithm for approximating observables. The multi-level machine learning algorithm is presented in section 3 and its application to uncertainty quantification (UQ) is presented in section 4. In section 5, we present extensions of the multi-level algorithm and discuss its implementation. Numerical experiments are presented in section 6.

2 The deep learning algorithm

2.1 Problem formulation

Our objective is to approximate observables with machine learning algorithms. For definiteness, we assume that the observable of interest is defined in terms of the solutions of the following very generic system of time-dependent parametric PDEs:

$$
\begin{aligned}
\partial_t \mathbf{U}(t, x, y) &= L\left(y, \mathbf{U}, \nabla_x \mathbf{U}, \nabla_x^2 \mathbf{U}, \dots\right), \quad \forall\, (t, x, y) \in [0, T] \times D(y) \times Y, \\
\mathbf{U}(0, x, y) &= \overline{\mathbf{U}}(x, y), \quad \forall\, (x, y) \in D(y) \times Y, \\
L_b \mathbf{U}(t, x, y) &= \mathbf{U}_b(t, x, y), \quad \forall\, (t, x, y) \in [0, T] \times \partial D(y) \times Y
\end{aligned}
\tag{2.1}
$$

Here, Y is the underlying parameter space and without loss of generality, we assume it to be $Y = [0, 1]^d$, for some $d \in \mathbb{N}$.

The spatial domain is labeled as $y \to D(y) \subset \mathbb{R}^{d_s}$ and $\mathbf{U} : [0, T] \times D \times Y \to \mathbb{R}^m$ is the vector of unknowns. The differential operator L is in a very generic form and can depend on the gradient and Hessian of $\mathbf{U}$, and possibly higher-order spatial derivatives. For instance, the heat equation as well as the Euler or Navier-Stokes equations of fluid dynamics are specific examples. Moreover, L_b is a generic operator for imposing boundary conditions.

The parametric nature of the PDE (2.1), represented by the parameter space Y, can stem from uncertainty quantification or Bayesian inversion problems where the parameter space models uncertainties in the PDE. The parametric nature can also arise from optimal design, control and PDE constrained

optimization problems with Y being the design (control) space. We equip this parameter space with a measure $\mu \in \mathrm{Prob}(Y)$.

For the parameterized PDE (2.1), we consider the following generic form of observables,

$$L_g(y, \mathbf{U}) := \int\limits_0^T \int\limits_{D_y} \psi(x, t) g(\mathbf{U}(t, x, y)) dx dt, \quad \text{for } \mu \text{ a.e } y \in Y. \tag{2.2}$$

Here, $\psi \in L^1_{\mathrm{loc}}(D_y \times (0, T))$ is a *test function* and $g \in C^s(\mathbb{R}^m)$, for $s \geqslant 1$.

For fixed functions ψ, g, we define the *parameters to observable* map:

$$\mathcal{L} : y \in Y \to \mathcal{L}(y) = L_g(y, \mathbf{U}), \tag{2.3}$$

with L_g being defined by (2.2).

We also assume that there exist suitable numerical schemes for approximating the PDE (2.1) for every parameter vector $y \in Y$, such that a high-resolution approximate solution $\mathbf{U}^\Delta(y) \approx \mathbf{U}(y)$ is available, with Δ denoting the grid resolution (mesh size, time step etc.). Hence, there exists an approximation to the *inputs to observable* map $\mathcal{L}^\Delta$ of the form,

$$\mathcal{L}^\Delta : y \in Y \to \mathcal{L}^\Delta(y) = L_g(y, \mathbf{U}^\Delta), \tag{2.4}$$

with the integrals in (2.2) being approximated to high accuracy by quadratures. Therefore, the original input parameters to observable map $\mathcal{L}$ is approximated by $\mathcal{L}^\Delta$ to very high accuracy i.e, for every value of a tolerance $\varepsilon > 0$, there exists a $\Delta \ll 1$, such that

$$\|\mathcal{L}(y) - \mathcal{L}^\Delta(y)\|_{L^p_\mu(Y)} < \varepsilon, \tag{2.5}$$

for some $1 \leqslant p \leqslant \infty$ and weighted norm,

$$\|f\|_{L^p_\mu(Y)} := \left(\int_Y |f(y)|^p d\mu(y) \right)^{\frac{1}{p}}.$$

The estimate (2.5) can be ensured by choosing Δ small enough in an underlying error estimate,

$$\|\mathcal{L}(y) - \mathcal{L}^\Delta(y)\|_{L^p_\mu(Y)} \sim \Delta^s, \tag{2.6}$$

for some $s > 0$.

2.2 Deep learning the parameters to observable map

The process of learning the (approximate) parameters to observable map $\mathcal{L}^\Delta$ (2.4) involves the following steps:

2.2.1 Training set.

As is customary in supervised learning ([10] and references therein), we need to generate or obtain data to train the network. To this end, we select a set of parameters $\mathcal{S} = \{y_i\}_{1 \leqslant i \leqslant N}$, with each $y_i \in Y$. The points in $\mathcal{S}$ can be chosen randomly from the parameter space Y, independently and identically distributed with the measure μ. Once the training set $\mathcal{S}$ is chosen, we perform a set of simulations of the underlying PDE (2.1), at a resolution Δ, to obtain $\mathcal{L}^\Delta(y)$, for all $y \in \mathcal{S}$.

2.2.2 Neural network.

Given an input vector $y \in Y$, a feedforward neural network (also termed as a multi-layer perceptron), shown in figure 1, transforms it to an output, through a layer of units (neurons) which compose of either affine-linear maps between units (in successive layers) or scalar non-linear activation functions within units [10], resulting in the representation,

$$\mathcal{L}_\theta(y) = C_K \circ \sigma \circ C_{K-1} \ldots\ldots\ldots \circ \sigma \circ C_2 \circ \sigma \circ C_1(y). \tag{2.7}$$
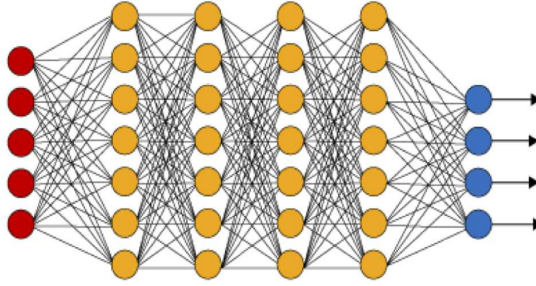
4

Figure 1: An illustration of a (fully connected) deep neural network. The red neurons represent the inputs to the network and the blue neurons denote the output layer. They are connected by hidden layers with yellow neurons. Each hidden unit (neuron) is connected by affine linear maps between units in different layers and then with nonlinear (scalar) activation functions within units.

Here, $\circ$ refers to the composition of functions and σ is a scalar (non-linear) activation function. A large variety of activation functions have been considered in the machine learning literature [10]. A very popular choice, which we will consider for the rest of this article, is the *ReLU* function,

$$\sigma(z) = \max(z, 0). \tag{2.8}$$

When, $z \in \mathbb{R}^p$ for some $p > 1$, then the output of the ReLU function in (2.8) is evaluated componentwise.

For any $1 \leqslant k \leqslant K$, we define

$$C_k z_k = W_k z_k + b_k, \quad \text{for } W_k \in \mathbb{R}^{d_{k+1} \times d_k}, z_k \in \mathbb{R}^{d_k}, b_k \in \mathbb{R}^{d_{k+1}}. \tag{2.9}$$

For consistency of notation, we set $d_1 = d$ and $d_K = 1$.

Thus in the terminology of machine learning (see also figure 1), our neural network (2.7) consists of an input layer, an output layer and $(K-1)$ hidden layers for some $1 < K \in \mathbb{N}$. The k-th hidden layer (with d_k neurons) is given an input vector $z_k \in \mathbb{R}^{d_k}$ and transforms it first by an affine linear map C_k (2.9) and then by a ReLU (or another) nonlinear (component wise) activation σ (2.8). A straightforward addition shows that our network contains $\left(d + 1 + \sum_{k=2}^{K-1} d_k\right)$ neurons. We also denote,

$$\theta = \{W_k, b_k\}, \theta_W = \{W_k\} \quad \forall \, 1 \leqslant k \leqslant K, \tag{2.10}$$

to be the concatenated set of (tunable) weights for our network. It is straightforward to check that $\theta \in \Theta \subset \mathbb{R}^M$ with

$$M = \sum_{k=1}^{K-1} (d_k + 1) d_{k+1}. \tag{2.11}$$

2.2.3 Loss functions and optimization.

For any $y \in \mathcal{S}$, one can readily compute the output of the neural network $\mathcal{L}_\theta(y)$ for any weight vector $\theta \in \Theta$. We define the so-called training *loss function* as

$$J(\theta) := \sum_{y \in \mathcal{S}} |\mathcal{L}^\Delta(y) - \mathcal{L}_\theta(y)|^p, \tag{2.12}$$

for some $1 \leqslant p < \infty$.

The goal of the training process in machine learning is to find the weight vector $\theta \in \Theta$, for which the loss function (2.12) is minimized.

It is common in machine learning [10] to regularize the minimization problem for the loss function i.e we seek to find,

$$\theta^* = \arg\min_{\theta \in \Theta} \left(J(\theta) + \lambda \mathcal{R}(\theta) \right). \tag{2.13}$$

Here, $\mathcal{R} : \Theta \to \mathbb{R}$ is a *regularization* (penalization) term. A popular choice is to set $\mathcal{R}(\theta) = \|\theta_W\|_q^q$ for either $q = 1$ (to induce sparsity) or $q = 2$. The parameter $0 \leqslant \lambda << 1$ balances the regularization term with the actual loss J (2.12).

The above minimization problem amounts to finding a minimum of a possibly non-convex function over a subset of $\mathbb{R}^M$ for possibly very large M. We follow standard practice in machine learning by either (approximately) solving (2.13) with a full-batch gradient descent algorithm or variants of mini-batch stochastic gradient descent (SGD) algorithms such as ADAM [15].

For notational simplicity, we denote the (approximate, local) minimum weight vector in (2.13) as θ^* and the underlying deep neural network $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ will be our neural network surrogate for the parameters to observable map $\mathcal{L}$ (2.4). The algorithm for computing this neural network is summarized below,

Algorithm 2.1. Deep learning of parameters to observable map.

Inputs: *Parameterized PDE* (2.1)*, Observable* (2.2)*, high-resolution numerical method for solving* (2.1) *and calculating* (2.2)*.*

Goal: *Find neural network $\mathcal{L}_{\theta^*}$ for approximating the parameters to observable map $\mathcal{L}$ (2.4).*

Step 1: *Choose the training set $\mathbb{S}$ and evaluate $\mathcal{L}^\Delta(y)$ for all $y \in \mathbb{S}$ by a numerical method.*

Step 2: *For an initial value of the weight vector $\overline{\theta} \in \Theta$, evaluate the neural network $\mathcal{L}_{\overline{\theta}}$ (2.7), the loss function (2.13) and its gradients to initialize the (stochastic) gradient descent algorithm.*

Step 3: *Run a stochastic gradient descent algorithm till an approximate local minimum θ^* of (2.13) is reached. The map $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ is the desired neural network approximating the parameters to observable map $\mathcal{L}$.*

2.3 An estimate on the generalization error of deep learning algorithm 2.1

For the rest of this section, we set $p = 1$ in (2.12) and aim to minimize the absolute value of the training loss, given by the training error,

$$\mathcal{E}_T = \mathcal{E}_T(\theta^*) := \frac{1}{N} \sum_{i=1}^{N} |\mathcal{L}^\Delta(y_i) - \mathcal{L}^*(y_i)|, \tag{2.14}$$

with $y_i \in \mathbb{S}$. The training error can be estimated from the loss function (2.12), a posteriori. Note that the training error $\mathcal{E}_T$ depends on the underlying randomly drawn training set $\mathbb{S}$ and, in general, is a random quantity. We suppress the y-dependence for notational convenience.

However, the relevant measure of the error within statistical learning theory is the so-called *generalization error* of the trained network, defined as

$$\mathcal{E}_G = \mathcal{E}_G(\theta^*) := \int_Y |\mathcal{L}^\Delta(y) - \mathcal{L}^*(y)| d\mu(y) \tag{2.15}$$

The task of estimating the generalization error (2.15) sharply, is one of the leading areas of investigation in theoretical machine learning [33, 1, 6] and references therein. In our setup, we can provide the following upper bound,

Theorem 2.2. *Assume that the training set $\mathbb{S} = \{y_i\}$ with $1 \leqslant i \leqslant N$ and each $y_i \in Y$ is independently and identically drawn from the underlying measure $\mu \in \mathrm{Prob}(Y)$, then the generalization gap i.e $\mathcal{E}_G - \mathcal{E}_T$, for the deep learning algorithm 2.1 for approximating the observable $\mathcal{L}^\Delta$ (2.4) is bounded (in the root mean square sense) by,*

$$\int_Y |\mathcal{E}_G - \mathcal{E}_T|^2 d\mu(y) \leqslant 4 \frac{\mathbb{V}\left(\mathcal{L}^\Delta\right) + \mathbb{V}\left(\mathcal{L}^*\right)}{N}. \tag{2.16}$$

Here, $\mathbb{V}(f)$ denotes the variance any underlying map $f : Y \to \mathbb{R}$,

$$\mathbb{V}(f) := \int\limits_Y (f(y))^2 \, d\mu(y) - \left(\int\limits_Y f(y)d\mu(y)\right)^2 \tag{2.17}$$

If the neural network (2.7) uses a ReLU activation function (2.8), a further (computable) upper bound on the generalization error is given by,

$$\int\limits_Y |\mathcal{E}_G - \mathcal{E}_T|^2 d\mu(y) \leqslant \frac{4\mathbb{V}\left(\mathcal{L}^\Delta\right) + \inf\limits_{1 \leqslant p \leqslant \infty} 2d^{\frac{1}{p}} \prod\limits_{k=1}^{K} \|W_k\|_p^2}{N}. \tag{2.18}$$

Here, $\|A\|_p$, for $1 \leqslant p \leqslant \infty$, is the ℓ_p matrix norm induced on any $m \times n$ matrix $A \in \mathbb{R}^{m \times n}$ by the ℓ_p norm on vectors in $\mathbb{R}^n$.

Proof. For any $y \in Y$, consider the integrand $I(y) = \mathcal{L}^\Delta(y) - \mathcal{L}^*(y)$. As the training points $y_i \in \mathcal{S}$ are randomly chosen from the underlying measure $\mu \in \text{Prob}(Y)$, we realize that the training error $\mathcal{E}_T$ (2.14) is the Monte Carlo quadrature approximation of the generalization error $\mathcal{E}_G$ (2.15) and we can estimate the difference between them, the so-called generalization gap, in terms of the central limit theorem [2] by,

$$\int\limits_Y |\mathcal{E}_G - \mathcal{E}_T|^2 d\mu(y) \leqslant \frac{\mathbb{V}(I)}{N}.$$

We estimate the variance in (4.1) in the following manner,

$$\mathbb{V}(I) = \int\limits_Y \left(|\mathcal{L}^\Delta(y) - \mathcal{L}^*(y)| - \int\limits_Y |\mathcal{L}^\Delta(z) - \mathcal{L}^*(z)|d\mu(z)\right)^2 d\mu(y)$$

$$= \int\limits_Y \left(\int\limits_Y \left(|\mathcal{L}^\Delta(y) - \mathcal{L}^*(y)| - |\mathcal{L}^\Delta(z) - \mathcal{L}^*(z)|\right) d\mu(z)\right)^2 d\mu(y)$$

$$\leqslant \int\limits_Y \left(\int\limits_Y \left(|\mathcal{L}^\Delta(y) - \mathcal{L}^*(y) - (\mathcal{L}^\Delta(z) - \mathcal{L}^*(z))|\right) d\mu(z)\right)^2 d\mu(y) \text{ (by triangle inequality)}$$

$$\leqslant \int\limits_Y \left(\int\limits_Y |\mathcal{L}^\Delta(y) - \mathcal{L}^\Delta(z)|d\mu(z) + \int\limits_Y |\mathcal{L}^*(y) - \mathcal{L}^*(z)|d\mu(z)\right)^2 d\mu(y) \text{ (by triangle inequality)}$$

$$\leqslant 2\left(\underbrace{\int\limits_Y \int\limits_Y |\mathcal{L}^\Delta(y) - \mathcal{L}^\Delta(z)|^2 d\mu(z)d\mu(y)}_{I_1} + \underbrace{\int\limits_Y \int\limits_Y |\mathcal{L}^*(y) - \mathcal{L}^*(z)|^2 d\mu(z)d\mu(y)}_{I_2}\right) \text{ (by Cauchy} - \text{Schwartz)}$$

It is straightforward to obtain the following estimates,

$$I_1 \leqslant 2\mathbb{V}\left(\mathcal{L}^\Delta\right), \ I_2 \leqslant 2\mathbb{V}\left(\mathcal{L}^*\right),$$

resulting in the estimate (2.16).

Moreover, the term I_2 can be further estimated in a straightforward manner by evaluating the neural network (2.7) successively and using the fact that

$$|\sigma(y) - \sigma(z)| \leqslant \|y - z\|_p,$$

7

for any $1 \leqslant p \leqslant \infty$ and for the ReLU activation function σ given by (2.8), in terms of

$$
\begin{aligned}
I_2 &= \int_Y \int_Y |\mathcal{L}^*(y) - \mathcal{L}^*(z)|^2 d\mu(z) d\mu(y) \\
&\leqslant \int_Y \int_Y \prod_{k=1}^{K} \|W_k\|_p^2 \|y - z\|_p^2 d\mu(z) d\mu(y) \\
&\leqslant d^{\frac{1}{p}} \prod_{k=1}^{K} \|W_k\|_p^2.
\end{aligned}
$$

The last inequality follows from the fact that $\|y - z\|_p \leqslant d^{\frac{1}{p}}$ for any $y, z \in Y = [0,1]^d$.

Combining the estimates on I_1 and I_2 yields the desired upper bound (2.18).

$\square$

Taking a square root in (2.16),(3.2) yields inequalities,

$$
\begin{aligned}
\mathcal{E}_G &\sim \mathcal{E}_T + 2\frac{\text{std}(\mathcal{L}^\Delta) + \text{std}(\mathcal{L}^*)}{\sqrt{N}}, \\
&\sim \mathcal{E}_T + \frac{2\text{std}(\mathcal{L}^\Delta) + \inf_{1 \leqslant p \leqslant \infty} \sqrt{2} d^{\frac{1}{2p}} \prod_{k=1}^{K} \|W_k\|_p}{\sqrt{N}},
\end{aligned}
\tag{2.19}
$$

with std denoting the standard deviation i.e, the square roof of the variance (2.17). However, the above equation should always be interpreted rigorously in the root mean square sense (2.16),(2.18).

Remark 2.3. The above estimate (2.19) on the generalization gap is clearly an upper bound and is not necessarily sharp. In fact, the proof of Theorem 2.2 makes it clear that the bound could be a significant overestimate. As is well known [1, 6, 25], estimating the generalization error of neural networks sharply is a notoriously hard problem. Nevertheless, above upper bound (2.19) will suffice for our purposes in this paper. ∎

3 A multi-level Deep learning algorithm.

As mentioned in the introduction, the generalization error estimates (2.16) and (2.19) explain potential limitations of the deep learning algorithm 2.1. We assume that the training process yields a neural network of form (2.7) with the following desirable properties,

$$
\text{std}(\mathcal{L}^*) \sim \text{std}(\mathcal{L}^\Delta), \quad \mathcal{E}_T \leqslant \frac{\text{std}(\mathcal{L}^\Delta)}{\sqrt{N}}.
\tag{3.1}
$$

This assumption automatically implies that the generalization error scales as,

$$
\mathcal{E}_G \sim \frac{\text{std}(\mathcal{L}^\Delta)}{\sqrt{N}}.
\tag{3.2}
$$

While it might not be feasible to find a trained neural network that satisfies these desirable properties (3.1) in all cases, such assumptions can be readily checked a posteriori.

Even under these assumptions, as long as $\text{std}(\mathcal{L}^\Delta) \sim \mathcal{O}(1)$, we see from (3.2) that the generalization error $\mathcal{E}_G \sim \frac{1}{\sqrt{N}}$. Hence, for obtaining a generalization error of say 1 percent, we need 10^4 training samples. Generating such a large number of training samples might be prohibitively expensive, particularly for if the underlying parametric PDE (2.1) is in two or three space dimensions.

Our goal in this section is to propose a *multi-level* version of the deep learning algorithm 2.1. The basis of this algorithm is the observation that the underlying parameters to observable map $\mathcal{L}$ (2.3) can
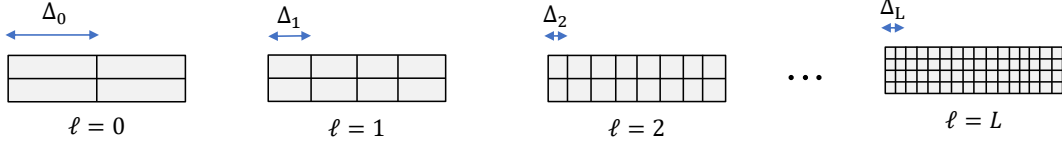
Figure 2: A schematic for a sequence of nested grids used in defining the multi-level algorithms.

be approximated on a sequence of mesh resolutions Δ_ℓ, for $0 \leqslant \ell \leqslant L$ for some $L > 0$. We require that $\Delta_\ell < \Delta_{\ell-1}$ for each $1 \leqslant \ell \leqslant L$, see Figure 2 for a diagrammatic representation of this sequence of grids.

For each ℓ, we assume that the underlying parameters to observable map $\mathcal{L}$ can be approximated by the map $\mathcal{L}^{\Delta_\ell}$, computed on resolution Δ_ℓ with an estimate,

$$\|\mathcal{L} - \mathcal{L}^{\Delta_\ell}\|_{L^1_\mu(Y)} \sim \Delta_\ell^s, \quad s > 0. \tag{3.3}$$

Given such a sequence of resolutions Δ_ℓ and approximate parameters to observable maps $\mathcal{L}^{\Delta_\ell}$, we have the following (telescopic decomposition)

$$\mathcal{L}^{\Delta_L}(y) = \mathcal{L}^{\Delta_0}(y) + \sum_{\ell=1}^{L} \left(\mathcal{L}^{\Delta_\ell}(y) - \mathcal{L}^{\Delta_{\ell-1}}(y) \right), \quad \forall y \in Y. \tag{3.4}$$

Introducing the so-called *details*,

$$\mathcal{D}_\ell(y) = \mathcal{L}^{\Delta_\ell}(y) - \mathcal{L}^{\Delta_{\ell-1}}(y), \quad \forall y \in Y, \ 1 \leqslant \ell \leqslant L, \tag{3.5}$$

we rewrite the telescopic decomposition (3.4) as

$$\mathcal{L}^{\Delta_L}(y) = \mathcal{L}^{\Delta_0}(y) + \sum_{\ell=1}^{L} \mathcal{D}_\ell(y), \quad \forall y \in Y. \tag{3.6}$$

The multi-level deep learning algorithm will be based on independently learning the following maps

$$\mathcal{L}^{\Delta_0}(y) \approx \mathcal{L}_0^*(y), \quad \mathcal{D}_\ell(y) \approx \mathcal{D}_\ell^*(y), \quad \forall y \in Y, \ 1 \leqslant \ell \leqslant L. \tag{3.7}$$

Here, $\mathcal{L}_0^*, \mathcal{D}_\ell^*$, for $1 \leqslant \ell \leqslant L$ are artificial neural networks of the form (2.7). The resulting algorithm is summarized below,

Algorithm 3.1. Multi-level deep learning of parameters to observable map.

Inputs: *Parameterized PDE (2.1), Observable (2.2), high-resolution numerical method for solving (2.1) and calculating (2.2), a sequence of grids with grid size Δ_ℓ for $0 \leqslant \ell \leqslant L$.*

Goal: *Find neural network $\mathcal{L}_{ml}^*$ for approximating the parameters to observable map $\mathcal{L}$ (2.3).*

Step 1: *For the coarsest mesh resolution Δ_0, select a training set $\mathbb{S}_0 = \{y_i^0\}$, $1 \leqslant i \leqslant N_0 = \#(\mathbb{S}_0)$, with $y_i^0 \in Y$ being independently and identically distributed with respect to the measure μ. For each y_i^0, compute $\mathcal{L}_0^\Delta(y_i^0)$ by solving the PDE (2.1) on a mesh resolution Δ_0 and computing the observable (2.2). With the training data $\{y_i^0, \mathcal{L}^{\Delta_0}(y_i^0)\}_{1 \leqslant i \leqslant N_0}$, find the neural network $\mathcal{L}_0^* \approx \mathcal{L}_0^\Delta$ by applying the deep learning algorithm 2.1.*

Step 2: *For each $1 \leqslant \ell \leqslant L$, select a training set $\mathbb{S}_\ell = \{y_i^\ell\}$, $1 \leqslant i \leqslant N_\ell = \#(\mathbb{S}_\ell)$, with $y_i^\ell \in Y$ being independently and identically distributed with respect to the measure μ. For each y_i^ℓ, compute $\mathcal{D}_\ell(y_i^\ell) = \mathcal{L}^{\Delta_\ell}(y_i^\ell) - \mathcal{L}^{\Delta_{\ell-1}}(y_i^\ell)$ by solving the PDE (2.1) on two mesh resolutions of mesh size Δ_ℓ and $\Delta_{\ell-1}$ and computing the observable (2.2). With the training data $\{y_i^\ell, \mathcal{D}_\ell(y_i^\ell)\}_{1 \leqslant i \leqslant N_\ell}$, find the neural network $\mathcal{D}_\ell^* \approx \mathcal{D}_\ell$ by applying the deep learning algorithm 2.1.*

Step 3*: Form the artificial neural network $\mathcal{L}_{ml}^* \approx \mathcal{L}$ by,*

$$\mathcal{L}_{ml}^*(y) = \mathcal{L}_0^*(y) + \sum_{\ell=1}^{L} \mathcal{D}_\ell^*(y), \quad \forall y \in Y. \tag{3.8}$$

The generalization error of the multi-level machine learning algorithm 3.1 is defined as,

$$\mathcal{E}_G^{ml} := \int\limits_Y |\mathcal{L}^{\Delta_L}(y) - \mathcal{L}_{ml}^*(y)| d\mu(y). \tag{3.9}$$

Implicitly, we have assumed that the approximation $\mathcal{L}^{\Delta_L}$ to the underlying parameters to observable map $\mathcal{L}$ at the finest level of resolution Δ_L is the *ground truth*.

We are going to estimate the generalization error (3.9) in terms of the generalization errors at the coarsest level of resolution and that of the details, defined as

$$\mathcal{E}_G^0 = \int\limits_Y |\mathcal{L}^{\Delta_0}(y) - \mathcal{L}_0^*(y)| d\mu(y), \quad \mathcal{E}_G^\ell = \int\limits_Y |\mathcal{D}_\ell(y) - \mathcal{D}_\ell^*(y)| d\mu(y), \quad 1 \leqslant \ell \leqslant L. \tag{3.10}$$

We have the following lemma for estimating the generalization error (3.9).

Lemma 3.2. *Assume that the training sets $\mathcal{S}_\ell = \{y_i^\ell\}$ with $1 \leqslant i \leqslant N_l$ and $0 \leqslant \ell \leqslant L$ and each $y_i^\ell \in Y$ is independently and identically drawn from the underlying measure $\mu \in \mathrm{Prob}(Y)$ and the neural networks use a ReLU activation function (2.8), then the generalization error (3.9) of the multi-level machine learning algorithm 3.1 is estimated as,*

$$\mathcal{E}_G^{ml} \leqslant \mathcal{E}_G^0 + \sum_{\ell=1}^{L} \mathcal{E}_G^\ell,$$

$$\int\limits_Y |\mathcal{E}_G^0 - \mathcal{E}_T^0|^2 d\mu(y) \leqslant 4 \frac{\mathbb{V}\left(\mathcal{L}^{\Delta_0}\right) + \mathbb{V}\left(\mathcal{L}_0^*\right)}{N_0}, \tag{3.11}$$

$$\int\limits_Y |\mathcal{E}_G^\ell - \mathcal{E}_T^\ell|^2 d\mu(y) \leqslant 4 \frac{\mathbb{V}\left(\mathcal{D}_\ell\right) + \mathbb{V}\left(\mathcal{D}_\ell^*\right)}{N_l}, \quad \text{for all} \quad 1 \leqslant \ell \leqslant L$$

Here, $\mathcal{E}_T^0, \mathcal{E}_T^\ell$ denote the training errors (2.14) obtained by applying the deep learning algorithm 2.1 to learn the maps $\mathcal{L}^{\Delta_0}, \mathcal{D}_\ell$, for all $1 \leqslant \ell \leqslant 1$.

The proof of the first inequality in (3.11) in above lemma is based on the telescopic decomposition (3.6) and successive applications of the triangle inequality. The other estimates in (3.11) follow from a direct application of the inequality (2.16) in Theorem 2.2 to the neural networks $\mathcal{L}_0^*, \mathcal{D}_\ell^*$, for all $1 \leqslant \ell \leqslant L$.

Formally, one can rewrite the estimate on generalization error as,

$$\mathcal{E}_G^{ml} \sim \mathcal{E}_T^0 + \sum_{\ell=1}^{L} \mathcal{E}_T^\ell + \frac{2(\mathrm{std}(\mathcal{L}^{\Delta_0}) + \mathrm{std}(\mathcal{L}_0^*))}{\sqrt{N_0}} + \sum_{\ell=1}^{L} \frac{2(\mathrm{std}(\mathcal{D}_\ell) + \mathrm{std}(\mathcal{D}_\ell^*))}{\sqrt{N_l}} \tag{3.12}$$

We need to determine the training sample numbers N_ℓ for $0 \leqslant \ell \leqslant L$, in order to complete the description of the multi-level machine learning algorithm 3.1. These have to be determined such that the multi-level deep learning algorithm 3.1 can lead to a greater accuracy (a lower generalization error $\mathcal{E}_G$) at a similar computational cost as the underlying deep learning algorithm 2.1.

A heuristic argument for selecting sample numbers runs as follows. First, we assume that the cost of training the neural networks and evaluating them is significantly lower than the cost of generating the training data with a PDE solver for (2.1). This assumption is indeed justified for most realistic problems (see table 13 in [19] for the training and evaluation costs vis a vis the cost of generating training data for a flow past airfoils). Next, we assume that the cost of solving a PDE such as (2.1) for a single realization

of the parameter vector $y \in Y$, on a mesh resolution of Δ_ℓ scales as $\Delta_\ell^{-(d_s+1)}$. This assumption is justified for first-order time-dependent PDEs (due to the CFL condition) and the arguments below can be readily extended to a more general case. It assume it here for the sake of definiteness.

Thus, it is much cheaper to generate a training sample on a coarse resolution than on a fine resolution. Hence, the intuitive idea is to generate a much larger number of training samples on coarser mesh resolutions than on finer resolutions (see Figure 2). From the generalization error estimate (3.12), we see that contribution to the generalization error from the details, corresponding to fine mesh resolutions, can be very low, even for a low number of training samples as long as the standard deviation of the underlying details is low. Thus, we can combine a large number of training samples at low resolution with a few training samples at high resolution, in order to obtain low generalization errors at comparable cost to the deep learning algorithm 2.1. These heuristic considerations are formalized in the lemma below.

Lemma 3.3. *For any given tolerance ε, there exists a $\Delta > 0$ such that the error estimate (2.5) holds, we assume that training process for the deep learning algorithm 2.1 results in a neural network $\mathcal{L}^*$ with properties (3.1). Moreover, we consider a generic sequence of mesh resolutions $s_n = \{\ell_k\}$ with $\ell_0 = 0$, $\ell_n = L$ that yields the following generalized formulation of the multi-level deep learning algorithm 3.1*

$$\mathcal{L}^*_{\ell_n}(y) = \mathcal{L}^*_{\ell_0}(y) + \sum_{k=1}^n \mathcal{D}^*_k(y), \quad \mathcal{L}^*_{\ell_0}(y) \approx \mathcal{L}^{\Delta_0}(y), \quad \mathcal{D}^*_k(y) \approx \mathcal{D}_k(y) := \mathcal{L}^{\Delta_{\ell_k}}(y) - \mathcal{L}^{\Delta_{\ell_{k-1}}}(y), \quad \forall y \in Y. \quad (3.13)$$

such that

$$\Delta_k = 2^{(\ell_n - \ell_k)} \Delta, \quad 0 \leqslant k \leqslant n. \quad (3.14)$$

and training sample numbers given by,

$$N_k \sim \frac{L \mathbb{V}_k}{\varepsilon^2}, \quad 0 \leqslant k \leqslant n, \quad \text{with } \mathbb{V}_0 = \mathbb{V}(\mathcal{L}^{\Delta_0}), \; \mathbb{V}_k = \mathbb{V}(\mathcal{D}_k), \quad 1 \leqslant k \leqslant n. \quad (3.15)$$

Furthermore, we assume that the training process in the multi-level deep learning algorithm 3.1 results in artificial neural networks $\mathcal{L}^*_0, \mathcal{D}^*_k$ with properties,

$$\mathcal{E}^0_T << \frac{\text{std}(\mathcal{L}^{\Delta_0})}{\sqrt{N_0}}, \; \text{std}(\mathcal{L}^*_0) \sim \text{std}(\mathcal{L}^{\Delta_0}), \; \mathcal{E}^k_T << \frac{\text{std}(\mathcal{D}_k)}{\sqrt{N_k}}, \; \text{std}(\mathcal{D}^*_k) \sim \text{std}(\mathcal{D}_k), \quad \forall 1 \leqslant k \leqslant n. \quad (3.16)$$

Let Σ_{ml} be speed up i.e, the ratio of computational cost of computing $\mathcal{L}^\Delta$ to accuracy of $O(\varepsilon)$. to the cost of computing $\mathcal{L}^\Delta$ to the same accuracy with the multi-level machine learning algorithm 3.1. Under the assumption that the costs of training and evaluation of all the neural networks in algorithms 2.1 and 3.1 is significantly smaller than the cost of generating the training data, we have the following estimate on Σ_{ml},

$$\frac{1}{\Sigma_{ml}} \sim \frac{L}{\mathbb{V}(L^\Delta)} \left[\mathbb{V}_0 2^{-L\bar{d}} + \sum_{k=1}^n \mathbb{V}_k 2^{-(L-\ell_k)\bar{d}} \right], \quad (3.17)$$

with $\bar{d} = d_s + 1$ is the number of space-time dimensions for the parametrized PDE (2.1).

Proof. Based on error estimate (2.6), we observe that $\varepsilon \sim \Delta^s$. Applying assumptions (3.1) on the neural network $\mathcal{L}^*$, generated by the deep learning algorithm 2.1, in the estimate (2.16) for the generalization error of $\mathcal{L}^*$, we obtain that

$$N \sim \mathbb{V}(\mathcal{L}^\Delta) \Delta^{-2s}, \quad (3.18)$$

training samples are required to approximate $\mathcal{L}^\Delta$ to tolerance ε.

Under our assumptions, the cost of training and evaluating neural networks is much smaller than the cost of generating the training data. Hence, the total cost of the deep learning algorithm 2.1 is given by,

$$\mathcal{C}_{DL} \sim N \mathcal{C}_\Delta \sim \mathbb{V}(\mathcal{L}^\Delta) \Delta^{-(2s+\bar{d})}. \quad (3.19)$$

Here, we have used (3.18) and the fact that the computational cost of generating a single training sample by solving the PDE (2.1) numerically at resolution Δ is given by $\mathcal{C}_\Delta \sim \Delta^{-\bar{d}}$

Next, we consider the multi-level machine learning algorithm 3.1. Applying the assumptions (3.14) on the resolutions, (3.15) on the training sample numbers and (3.16) on trained neural networks in the formula for the generalization error (3.11), we observe that each term in the right hand side of the upper bound in (3.11) scales as $\mathcal{O}\left(\frac{\varepsilon}{n}\right)$ and the total generalization error is of $\mathcal{O}(\varepsilon)$.

Under the assumptions that the cost of training and evaluating neural networks is much smaller than the cost of generating the training data, the total cost of the multi-level machine learning algorithm 3.1 is given by,

$$
\begin{aligned}
\mathcal{C}_{ML} &\sim \sum_{k=0}^{n} N_k \mathcal{C}_{\Delta_k} \\
&\sim \sum_{k=0}^{n} L \mathbb{V}_k 2^{-(L-\ell_k)\bar{d}} \Delta^{-(2s+\bar{d})}, \quad \text{by } (3.14),(3.15).
\end{aligned}
\tag{3.20}
$$

The estimate on speed-up (3.17) follows from dividing (3.20) by (3.19).

$\square$

Remark 3.4. Each sequence of resolutions in the multi-level method can be specified in terms of a single parameter, i.e *model complexity* defined as:

$$
c_{ml} = \frac{n^2}{L}
\tag{3.21}
$$

∎

Remark 3.5. From formula (3.17), we see that speedup of the multi-level algorithm 3.1 over the underlying machine learning algorithm 2.1 is expressed in terms of the *variance of the details* $\mathcal{D}_\ell$. In practice, $\mathbb{V}_0 = \mathbb{V}(\mathcal{L}^{\Delta_0}) \sim \mathbb{V}(\mathcal{L}^{\Delta})$. On the other hand, a straightforward application of the error estimate (2.6) yields the following estimate,

$$
\mathbb{V}_k = \mathbb{V}(\mathcal{D}_k) = \mathbb{V}(\mathcal{L}^{\Delta_{\ell_k}} - \mathcal{L}^{\Delta_{\ell_{k-1}}}) \sim \Delta_{\ell_{k-1}}^{2s} = \Delta^{2s} 2^{2(L-\ell_{k-1})s}, \quad \text{from } (3.14).
\tag{3.22}
$$

Assuming for simplicity that $\ell_k = k\frac{L}{n} \ \forall k : 1 \leqslant k \leqslant n$ and substituting (3.22) in (3.17) results in

$$
\frac{1}{\Sigma_{ml}} \sim L 2^{-L\bar{d}} + \frac{L\Delta^{2s} 2^{2s\frac{L}{n}}}{\mathbb{V}(\mathcal{L}^{\Delta})} \sum_{k=1}^{n} 2^{L(1-\frac{k}{n})(2s-\bar{d})}
\tag{3.23}
$$

The geometric series in (3.23) clearly converges to a finite value as long as $s \leqslant \frac{\bar{d}}{2}$. This holds true is most cases of practical interest as $s \leqslant 1$ and $\bar{d} = 3$ or 4. Consequently, the speedup will be exponential in L for small values of Δ. ∎

Remark 3.6. We have used a very crude strategy of selecting sample numbers in (3.15). It relies on specifying the number of samples at the finest level N_L and at the coarsest level N_0. The samples at intermediate levels are determined by,

$$
N_k = N_L 2^{e(L-\ell_k)}, \ 0 < k < n, \quad e = \frac{\log_2(N_0/N_L)}{L}.
\tag{3.24}
$$

More sophisticated strategies, such as those proposed in the context of multi-level Monte Carlo methods in [9] and references therein, might lead to greater speedup with the multi-level machine learning algorithms. ∎

4 Uncertainty Quantification

In this article, we focus our attention on forward uncertainty quantification (UQ) or uncertainty propagation with respect to the parameters to observables map $\mathcal{L}$ (2.3) (or rather its numerical surrogate $\mathcal{L}^{\Delta}$ (2.4)). To this end, we follow [19] and consider the so-called *push forward measure* with respect to this map, i.e, $\hat{\mu}^{\Delta} \in \text{Prob}(\mathbb{R})$ given by

$$
\hat{\mu}^{\Delta} := \mathcal{L}^{\Delta} \# \mu, \quad \Rightarrow \quad \int_{\mathbb{R}} f(z) d\hat{\mu}^{\Delta}(z) = \int_{Y} f(\mathcal{L}^{\Delta}(y)) d\mu(y),
\tag{4.1}
$$

for any μ-measurable function $f : \mathbb{R} \to \mathbb{R}$.

Note that the measure $\hat{\mu}^\Delta$ contains all the statistical information about the map $\mathcal{L}^\Delta$. In particular, any moment or statistical quantity of interest with respect to this map can be computed by choosing a suitable test function f in (4.1). In particular, choosing $f(z) = z$ provides the mean $\bar{\mathcal{L}}$ of the observable $\mathcal{L}^\Delta$ and $f(z) = (z - \bar{\mathcal{L}})^2$ yields the variance.

The baseline Monte Carlo algorithm for approximating this measure (probability distribution) consists of choosing J independent, identically distributed (with respect to μ) samples $y_j \in Y$ and approximating the measure $\hat{\mu}^\Delta$ by the so-called *empirical measure*,

$$\hat{\mu}_{mc} = \frac{1}{J}\sum_{j=1}^{J} \delta_{\mathcal{L}^\Delta(y_j)} \quad \Rightarrow \quad \int_{\mathbb{R}} f(z)d\hat{\mu}_{qmc}(z) = \frac{1}{J}\sum_{j=1}^{J} f\left(\mathcal{L}^\Delta(y_j)\right). \tag{4.2}$$

On the other hand, within the *deep learning Monte Carlo* (DLMC) algorithm, proposed in [19], we first generate a deep neural network $\mathcal{L}^*$ to approximate the underlying map $\mathcal{L}^\Delta$ by the deep learning algorithm 2.1 and then approximate the measure $\hat{\mu}^\Delta$ by,

$$\hat{\mu}_{mc}^* = \frac{1}{J_L}\sum_{j=1}^{J_L} \delta_{\mathcal{L}^*(y_j)}, \tag{4.3}$$

with $J_L \gg J$ evaluations of the neural network. In [19], the authors provide a complexity analysis of the DLMC algorithm and also estimate the speedup with respect to the baseline Monte Carlo algorithm, in terms of the errors of computing the measure $\hat{\mu}^\Delta$ in the Wasserstein metric. We refer the interested reader to [19], Theorem 3.11. The numerical experiments, presented in [19] also provide evidence of a significant speedup with the DLMC algorithm over the baseline Monte Carlo method.

Here, we propose another variant of the DLMC algorithm, which is based on the multi-level Machine learning algorithm 3.1. The algorithm is as follows,

Algorithm 4.1. A Multi-level Machine learning Monte Carlo (ML2MC) algorithm for forward UQ.

Inputs: *Parameterized PDE* (2.1), *Observable* (2.2), *high-resolution numerical method for solving* (2.1) *and calculating* (2.2), *a sequence of grids with grid size* Δ_ℓ *for* $0 \leqslant \ell \leqslant L$.

Goal: *Find a measure* $\hat{\mu}_{ml2mc} \in \mathrm{Prob}(\mathbb{R})$ *to approximate the push-forward measure* $\hat{\mu}^\Delta$ (4.1).

Step 1: *Generate the neural network* $\mathcal{L}_{ml}^* \approx \mathcal{L}^\Delta$ *by applying the multi-level machine learning algorithm 3.1.*

Step 2: *Define the approximate push-forward measure by,*

$$\hat{\mu}_{ml2mc}^* = \frac{1}{J_L}\sum_{j=1}^{J_L} \delta_{\mathcal{L}_{ml}^*(y_j)}, \tag{4.4}$$

One can readily perform a complexity analysis, completely analogous to section 3.3 of [19] to quantify possible speedups with the ML2MC algorithm over the DLMC algorithm by combining the arguments in Theorem 3.11 of [19], with the speedup estimate (3.17).

5 Extensions and Implementation

The multi-level machine learning algorithm 3.1 can be readily extended in the following directions,

5.1 Quasi-Monte Carlo type algorithms

In the recent paper [19], the authors obtained significantly lower generalization errors by choosing *low-discrepancy sequences*, instead of randomly distributed points, as the training set $\mathcal{S}$ in Step 1 of the deep learning algorithm 2.1. The intuitive reason for this was the fact that low-discrepancy sequences,

such as the Sobol and Halton points popularly used in Quasi-Monte Carlo integration methods [2], are equi-distributed in the underlying parametric domain Y. We can readily adapt the multi-level algorithm 3.1 to this setting by requiring that the training sets S_ℓ for $0 \leqslant \ell \leqslant L$ are chosen as consecutive Sobol (or Halton) points. The rest of the algorithm is unchanged. Similarly, the UQ algorithm 4.1 can be readily adapted to this context.

5.2 Other surrogate models.

Deep neural networks are only one possible machine learning surrogate for the parameters to observable map $\mathcal{L}$ (2.4). Another popular class of surrogate models are *Gaussian process regressions* [29], which belong to a larger class of so-called Bayesian models. Gaussian process regressions (GPR) rely on the assumption that the underlying map Δ is drawn from a Gaussin measure on a suitable function space, parameterized by,

$$\mathcal{L}(y) \sim GP(m(y), k(y, y')),\tag{5.1}$$

Here, $m(y) = \mathbb{E}[\mathcal{L}(y)]$ is the mean and $k(y, y') = \mathbb{E}[(\mathcal{L}(y) - m(y))(\mathcal{L}(y') - m(y'))]$ is the underlying covariance function. The mean and the covariance parametrize the so-called *prior* measure. It is common to assume that $m \equiv 0$

Given a training set $Y \supset S = \{y_i\}, 1 \leqslant i \leqslant n$, the key idea underlying a Gaussian process regression is to apply Bayes theorem and update the conditional distribution for a test set $Y_* \subset Y$, $Y_* \cap S = \Phi$ with $\#(Y_*) = n_*$. For any $y_* \in Y_*$, denote $z_* = \mathcal{L}(y_*)$, then one uses the Gaussian nature of the distributions to calculate the *posterior* conditional probability by the formula,

$$\text{Prob}(z_*|y_*, S) \sim \mathcal{N}\left(G_*^T G^{-1} z, \quad G_{**} - G_*^T G^{-1} G_*\right),\tag{5.2}$$

with $\mathcal{N}$ denoting a Gaussian distribution and $z = [\mathcal{L}(y_1), \ldots, \mathcal{L}(y_n)]$, $y_i \in S$. Here, $G \in \mathbb{R}^{n \times n}$, $G_* \in \mathbb{R}^{n \times n_*}$ and $G_{**} \in \mathbb{R}^{n_* \times n_*}$ are the training, the training-test and test Gram matrices, respectively, given by,

$$G^{(i,j)} = k(y_i, y_j), \quad G_*^{(i,j)} = k(y_i, y_{*,j}), \quad \text{and} \quad G_{**}^{(i,j)} = k(y_{*,i}, y_{*,j}).\tag{5.3}$$

Thus, computation of the conditional probability (5.2) requires the inversion of the training Gram matrix G (5.3), for instance by a Cholesky algorithm, entailing a computational cost of $O(n^3)$.

Popular choices for the covariance function in (5.1) are the squared exponential (RBF function) and Matern covariance functions,

$$k_{SE}(y, y') = \exp\left(-\frac{||y - y'||^2}{2\ell^2}\right), \quad k_{Matern}(y, y') = \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\sqrt{2\nu}\frac{||y - y'||}{\ell}\right)^\nu K_\nu\left(\sqrt{2\nu}\frac{||y - y'||}{\ell}\right).\tag{5.4}$$

Here $||\cdot||$ denotes the standard euclidean norm, K_ν is the Bessel function and ℓ the *characteristic length*, describing the length scale of the correlations between the points y and y'.

We can readily adapt the multi-level algorithm 3.1 to other surrogate models such as Gaussian process regressions. In fact, we propose a significantly more general form of the algorithm 3.1 below,

Algorithm 5.1. Multi-level learning of parameters to observable map.

Inputs: *Parameterized PDE (2.1), Observable (2.2), high-resolution numerical method for solving (2.1) and calculating (2.2), sequence of mesh resolutions s_n, number of training samples N_0, number of training samples N_L.*

Goal: *Compute a machine learning surrogate $\mathcal{L}_{ml}^*$ for approximating the parameters to observable map $\mathcal{L}$ (2.3).*

Step 1: *For the coarsest mesh resolution ℓ_0, select a training set $S_0 = \{y_i^0\}$, $1 \leqslant i \leqslant N_0 = \#(S_0)$, constituted by either random i.i.d points or with consecutive low-discrepency sequences such as Sobol points. For each y_i^0, compute $\mathcal{L}_0^\Delta(y_i^0)$ by solving the PDE (2.1) on a mesh resolution Δ_0 and computing the observable (2.2). With the training data $\{y_i^0, \mathcal{L}^{\Delta_0}(y_i^0)\}_{1 \leqslant i \leqslant N_0}$, train the neural network $\mathcal{L}_{0,NN}^* \approx \mathcal{L}_0^\Delta$ by algorithm 2.1 and the Gaussian Process regressor $\mathcal{L}_{0,GP}^* \approx \mathcal{L}_0^\Delta$, with suitable choice of the model hyperparameters. Assemble the ensemble model $\mathcal{L}_0^* = \alpha_{NN}^0 \mathcal{L}_{0,NN}^* + \alpha_{GP}^0 \mathcal{L}_{0,GP}^*$*

Step 2: *For each $\ell_k \in s_n$, $1 \leqslant k \leqslant n$, select a training set $\mathcal{S}_k = \{y_i^{\ell_k}\}$, $1 \leqslant i \leqslant N_k = \#(\mathcal{S}_k)$, with N_k defined as in (3.24) and $\mathcal{S}_k$ consisting of either random points or Sobol points. For each $y_i^{\ell_k}$, compute $\mathcal{D}_k(y_i^{\ell_k}) = \mathcal{L}^{\Delta_{\ell_k}}(y_i^{\ell_k}) - \mathcal{L}^{\Delta_{\ell_{k-1}}}(y_i^{\ell_k})$ by solving the PDE (2.1) on two successive mesh resolutions of Δ_{ℓ_k} and $\Delta_{\ell_{k-1}}$ and computing the observable (2.2). With the training data $\{y_i^{\ell_k}, \mathcal{D}_k(y_i^{\ell_k})\}_{1 \leqslant i \leqslant N_k}$ train the neural network $\mathcal{D}_{k,NN}^* \approx \mathcal{D}_k$ and the Gaussian Process regressor $\mathcal{D}_{k,GP}^* \approx \mathcal{D}_k$, with suitable choice of the model hyperparameters. Assemble the ensemble model $\mathcal{D}_k^* = \alpha_{NN}^k \mathcal{D}_{k,NN}^* + \alpha_{GP}^k \mathcal{D}_{k,GP}^*$*

Step 3: *Form the machine learning surrogate $\mathcal{L}_{ml}^* \approx \mathcal{L}^{\Delta_L}$ as*

$$\mathcal{L}_{ml}^*(y) = \mathcal{L}_0^*(y) + \sum_{k=1}^k \mathcal{D}_k^*(y), \quad \forall y \in Y. \tag{5.5}$$

5.3 Selection of hyperparameters.

There are quite a few hyperparameters in the multi-level algorithm 5.1. We choose these hyperparameters with the following procedure,

5.3.1 Choice of Neural network hyperparameters

The deep learning algorithm 2.1 requires specification of the following hyperparameters: the architecture of the neural network (2.7) (number of layers (depth) and size of each layer (width), the exponent p in the loss function (2.12), the exponent q and constant λ in the regularization term (2.13), the choice of the optimization algorithm for minimizing (2.13) and the starting value for it.

Following [19], we consider either $p = 1$ or $p = 2$ in the loss function (2.12). Similarly, the ADAM version of stochastic gradient algorithm [15] is used in *full batch* mode with a learning rate of $\eta = 0.01$ and is terminated at 10000 epochs. For the regularization terms and starting value for ADAM, we use the *ensemble training procedure* of [19] with either $q = 1$ or $q = 2$ and $\lambda = 5 \times 10^{-7}, 10^{-6}, 5 \times 10^{-6}, 10^{-5}, 5 \times 10^{-5}, 10^{-4}$. Similarly, five starting values (based on the He initialization) are used. Once the ensemble is trained, we select the hyperparameters that correspond to the smallest validation error, calculated on a validation set, created by setting aside 10% of the training samples.

5.3.2 Choice of Gaussian process hyperparameters.

For the characteristic length l in (5.3), we minimize the log negative marginal likelihood [29]. This leaves two hyperparameters i.e, the choice of the covariance kernel in (5.2) and if we choose the Matern kernel (5.3), the choice of the parameter ν. Here, we consider four hyperparameters configurations for the Gaussian process regression, namely either the squared exponential kernel in (5.3) or the Matern kernel with $\nu = 0.5, 1.5, 2.5$. These parameters are determined from an ensemble training process by selecting the parameter with the least validation error.

5.3.3 Choice of ensemble model coefficients.

If $N_k > 500$, $0 \leqslant k \leqslant n$, we determine the ensemble hyperparameters $\alpha_{NN,GP}^k$ in algorithm 5.1 by performing a linear least squares regression on the ensemble model with respect to a validation set, accounting for 10% of the training set. Otherwise, we simply set $\alpha_{GP}^k = \alpha_{NN}^k = 0.5$.

5.3.4 Choice of multi-level hyperparameters

In addition to the above hyperparameters, the multi-level algorithms 3.1 and 5.1 involve three additional hyperparameters i.e, the sequence $s_n = \{\ell_k\}_{k=1}^n$ of mesh resolutions, parametrized by the model complexity c_{ml} (3.21), the number of samples N_0 at the coarsest level to learn $\mathcal{L}^{\Delta_0}$ and the number of samples N_L to learn the detail $\mathcal{D}_n = \mathcal{L}^{\Delta_{\ell_n}} - \mathcal{L}^{\Delta_{\ell_{n-1}}}$, at the finest resolution. We will perform a sensitivity study to assess influence of these hyperparameters on the quality of results.
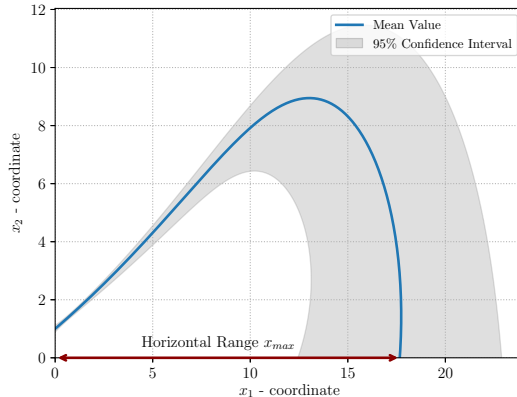
Figure 3: An illustration of two-dimensional projectile motion, with the mean value and the envelope of trajectories, corresponding to the 95% interval shown.

6 Numerical Experiments.

6.1 Projectile motion.

We start with a dynamical system modeling the motion of a projectile, subjected to both gravity as well as air drag and described by the nonlinear system of ODEs,

$$\frac{d}{dt}\boldsymbol{x}(t;y) = \boldsymbol{v}(t;y), \quad \frac{d}{dt}\boldsymbol{v}(t;y) = -F_D\big(\boldsymbol{v}(t;y);y\big)\boldsymbol{e}_1 - g\boldsymbol{e}_2$$
$$\boldsymbol{x}(y;0) = \boldsymbol{x}_0(y), \quad \frac{d\boldsymbol{x}(y;0)}{dt} = \boldsymbol{v}_0(y). \tag{6.1}$$

Here, $F_D = \frac{1}{2m}\rho C_d \pi r^2 ||\boldsymbol{v}||^2$ denotes the drag force and $\boldsymbol{x}_0(y) = [0, \ h]$, $\boldsymbol{v}_0(y) = [v_0 \cos(\alpha), \ v_0 \sin(\alpha)]$ are the initial position and velocity of the object, see figure 3 for a schematic representation.

On account of measurement errors, the system is described by the following uncertain parameters,

$$\rho(y) = 1.225\big(1 + \varepsilon G_1(y)\big), \ r(y) = 0.23\big(1 + \varepsilon G_2(y)\big), \ C_D(y) = 0.1\big(1 + \varepsilon G_3(y)\big)$$
$$m(y) = 0.145\big(1 + \varepsilon G_4(y)\big), \ h(y) = \big(1 + \varepsilon G_5(y)\big), \ \alpha(y) = 30°\big(1 + \varepsilon G_6(y)\big), \ v_0(y) = 25\big(1 + \varepsilon G_7(y)\big). \tag{6.2}$$

Here, $y \in [0,1]^7$ describes the input parameter space, with uniform distribution, and $G_k(y) = 2y_k - 1$ for $k = 1, ..., 7$, with $\varepsilon = 0.1$. The objective of the simulation is to compute and quantify uncertainty with respect to the observable corresponding to the *horizontal range* x_{max} (see figure 3),

$$\mathcal{L}(y) = x_{max}(y) = x_1(y; t_f), \quad \text{with } t_f = x_2^{-1}(0). \tag{6.3}$$

Although at first glance, this problem appears simplistic, it has all the features of much more complicated problems (such as the one considered in the next experiment). Namely, the input parameter space Y is moderately high-dimensional with 7 dimensions and the parameters to the observable map is highly non-linear. Moreover, it yields a significant amount of variance in the trajectories and the observable (see figure 3). Therefore, we will approximate solutions with a forward Euler discretization of (6.1). The main advantage for choosing this model lies in the fact that we can compute solutions of (6.1) for a very large number of samples (realizations of y in (6.2)) with minimal computational work. This allows us to compute reference solutions and test our algorithms carefully.

6.1.1 Upper bounds on the generalization error.

In section 2.3, theorem 2.2, we provided an upper bound (2.16), (2.19), on the generalization error for the deep learning algorithm 2.1. Our subsequent theory about the utility of multi-level training, rested

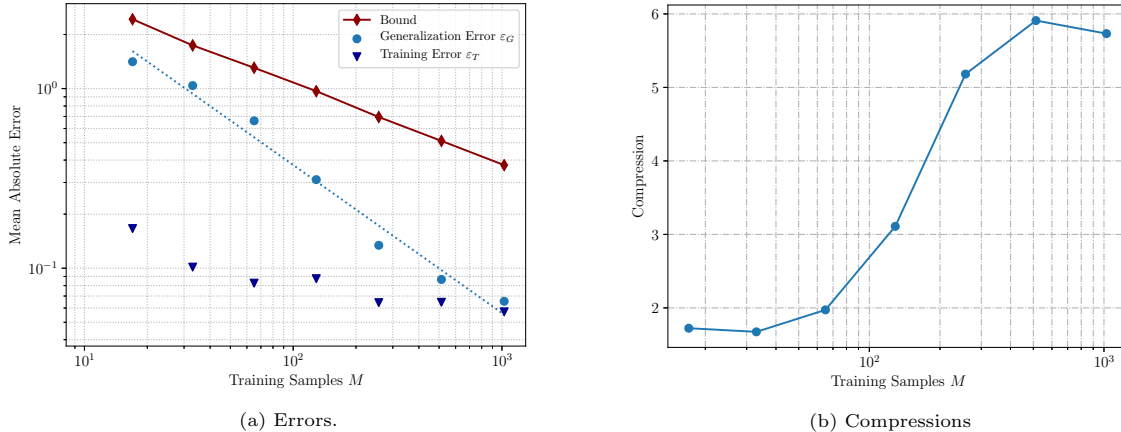(a) Errors.   (b) Compressions

Figure 4: Errors for the projectile motion. Left: Number of samples (X-axis) vs. training error, generalization error and computable upper bound (2.19) on the generalization error. Right: Compression (Ratio of the upper bound to the generalization error) (Y-axis) vs. Number of training samples (X-axis).

partially on this upper bound. We test our algorithm to check how sharp this upper bound is for this particular problem.

To this end, we fix a neural network with 6 hidden layers and 10 neurons per layer, resulting in a network with 638 tunable parameters (weights and biases). A mean absolute error loss function with a mean square regularization (with $\lambda = 10^{-6}$ in (2.13)) is minimized with the ADAM optimizer for a fixed learning rate of $\eta = 0.01$. To test the upper bound (2.19), we choose training sets as independent, uniformly distributed points in $[0,1]^7$ and the observable $\mathcal{L}$ (6.3) is computed with a forward Euler discretization with time step $\Delta t = 0.00125$. We trained the above neural network with training sets of size $N_r = 2^r, 4 \leqslant r \leqslant 10$ and computed the generalization error with respect to $2000 - N_r$ i.i.d points in $[0,1]^7$. Given the probabilistic nature of the upper bound (2.19), we retrain the network $K = 60$ times and compute the corresponding root mean square errors. As the right-hand side of (2.19) involves computing standard deviations, we estimate $\text{std}(\mathcal{L})$ from 2000 samples of $\mathcal{L}$ and $\text{std}(\mathcal{L}^*)$ from 1000 realizations of the neural network.

The results are presented in figure 4. In figure 4 (left), we plot the training error, estimated generalization error and computed upper bound (rhs of (2.19) (first line)) versus the number of training samples. We see from this figure, that the training error is significantly lower than the estimated generalization error, at least for a low number of training samples. The generalization error decays as $N^{-0.82}$, whereas the upper bound (2.19) provides a reasonable overestimate of this generalization error. To further quantify the overestimate, we plot the *compression* i.e, the ratio of the upper bound (2.19) to the computed generalization error in figure 4 (right) and find that it lies between a factor of 2 and 6. This sharpness of the upper bound is particularly impressive given how difficult it is to obtain sharp upper bounds for neural networks [1, 6] and references therein.

6.1.2 Results of multi-level training and uncertainty quantification.

Given the sharpness of the bound (2.19) on the generalization error and the fact that the training error seems to be significantly less than the generalization error, we can follow Lemma 3.3 and expect that the multi-level machine learning algorithm 3.1 will provide a lower generalization error than the deep learning algorithm 2.1 for this problem. To test this, we consider 7 successive time step resolutions $(L = 6)$ within algorithm 3.1. Moreover, $\Delta t_0 = 0.08$ at coarsest resolution and $\Delta t_L = 0.00125$ at the finest resolution. We performed an ensemble training for the selection of the exponent q of the regularization term and the parameter λ of $\mathcal{L}_0^*$ and $\mathcal{D}_{ref}^*$ (which the detail corresponding to the two coarsest resolutions). The remaining parameters are identical to the previous subsection, with exception of the loss function, that in this case is the mean squared error $(p = 2$ in (2.12)). L^2 regularization with $\lambda = 5 \times 10^{-7}$ revealed the

17

(a) Prediction error vs. Computational cost

(b) Gains (6.5)
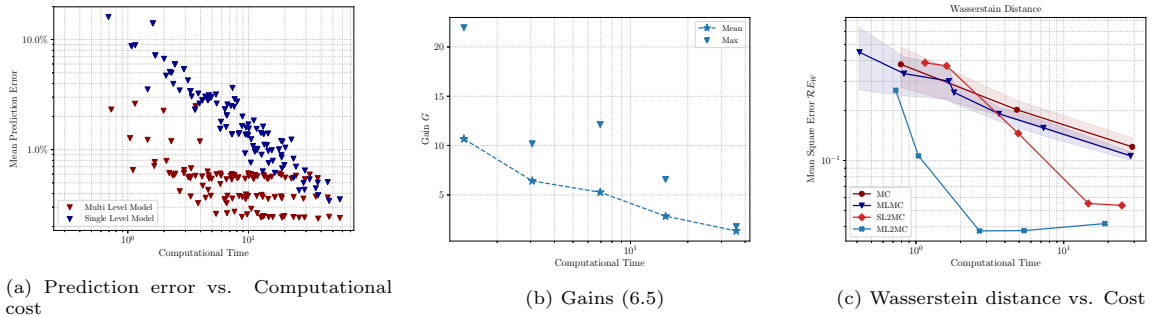
(c) Wasserstein distance vs. Cost

Figure 5: Results for the projectile motion case. Left: Prediction errors (Y-axis) vs computational cost (X-axis) for different multi-level parameters in algorithm 3.1 and corresponding results with deep learning algorithm 2.1. Center: Gain (6.5) vs. cost. Right: Errors, measured in the Wasserstein distance for the Monte-Carlo (MC), multi-level Monte Carlo (MLMC), single level machine learning (SL2MC) and multi-level machine learning algorithms (ML2MC) vs. Computational cost.

best performing configuration for both the maps. The same set of parameters found for $\mathcal{D}^*_{ref}$ was used for all the details. Moreover, we consider the following set of multi-level hyperparameters,

- 4 sequences of multi-level resolutions corresponding to $s_n^1 = \{0, 6\}$ ($c_{ml} = 0.16$), $s_n^2 = \{0, 3, 6\}$ ($c_{ml} = 0.67$), $s_n^3 = \{0, 2, 4, 6\}$, $c_{ml} = 1.5$, $s_n^4 = \{0, 1, 2, 3, 4, 5, 6\}$ ($c_{ml} = 6.0$).

- 4 different choices of number of training samples at the coarsest resolution $N_0 = \{256, 512, 1024, 2048\}$

- 7 different choices of number of training samples at the highest resolution $N_L = \{4, 8, 16, 32, 64, 92, 128\}$

Overall, we have 112 multi-level configurations. For each configuration, we run the multi-level algorithm 3.1 and compute the generalization error from only high-fidelity samples i.e, samples at the finest mesh resolution. As we constrain the total number of high-fidelity samples to 2000, we approximate the generalization error (2.15) with the prediction error $\mathcal{E}$,

$$\mathcal{E} := \frac{1}{N_\mathcal{T}} \left(\sum_{y \in \mathcal{T}} \left| \mathcal{L}^{\Delta_L}(y) - \mathcal{L}^*(y) \right|^p \right)^{\frac{1}{p}}, \tag{6.4}$$

by choosing test sets $\mathcal{T} \subset Y$ with number of samples $N_\mathcal{T} = \#(\mathcal{T})$ ranging from 1872 to 1996. We choose $p = 2$ in this section. For the sake of comparison, the deep learning algorithm 2.1 is run on this data with the number of training samples determined by the need to match the cost of the multi-level algorithm 3.1 (approximately).

The corresponding results for the generalization error are plotted in figure 5 (Left). In this figure, we plot the generalization error for each multi-level hyperparameter configuration (and the corresponding equally expensive deep learning algorithm) versus the total computational time (in seconds). From this figure, we observe that the error with deep learning algorithm reduces with time (number of training samples). Moreover, the multi-level algorithms lead to a significant reduction in error at the same computational cost to the deep learning algorithm 2.1. A close inspection of this figure reveals that about 95% of multi-level hyperparameters resulted in an error reduction (at the same cost) over the deep learning algorithm 2.1 and about 7% led to an order of magnitude reduction in error.

This gain in efficiency is quantified in terms of the so-called *gain*:

$$G = \frac{\mathcal{E}_{dl}}{\mathcal{E}_{ml}}, \tag{6.5}$$

with $\mathcal{E}_{dl,ml}$ corresponding to the *prediction errors* of the the deep learning 2.1 and multi-level 3.1 algorithms, respectively. We plot the mean and the maximum value of the gain G (6.5) with respect to multi-level hyperparameters for a range of computational costs, in figure 5 (Center). From this figure, we see a mean gain between 2 and 8 and a maximum gain of 12 for the multi-level algorithm over the

deep learning algorithm 2.1. Larger gains were obtained for lower computational costs (less number of training samples), which is the case of practical interest.

Finally, we apply the multi-level algorithm 3.1 in the context of uncertainty quantification, in the form of the ML2MC algorithm 4.1. We compute approximations $\hat{\mu}^*_{ml2mc}$ to the full push-forward measure $\hat{\mu}^\Delta$ (4.1). A reference push-forward measure $\hat{\mu}^{ref}$ is computed with a very small time step of $\Delta t = 0.001$ and 20000 Monte Carlo samples and we ascertain the quality of the ML2MC algorithm by computing *Wasserstein distances* $\mathcal{W}_1\left(\hat{\mu}^*_{ml2mc}, \hat{\mu}^{ref}\right)$ with the function *wasserstein_distance* of the Python library *scipy.stats* [38], on the multi-level hyperparameter configurations shown in Table 1. These hyperparameters approximately correspond to those that result in the highest gain in the prediction error (see figure 5 (A)).

In order to compare the multi-level UQ algorithm 4.1 with existing algorithms, we select the following,

- Standard Monte Carlo approximation of the push forward measure, at the finest resolution of $\Delta t = 0.00125$.

- A multi-level Monte Carlo algorithm for computing push-forward measures as proposed in [20].

- The *single-level* variant (SL2MC) of algorithm 4.1 where the multi-level algorithm 3.1 is replaced by the deep learning algorithm 2.1, trained with 9, 15, 21, 64, 191 and 322 number of samples at the finest mesh resolution. This single-level algorithm coincides with the DLMC algorithm of [19].

In figure 5 (Right), we plot the mean Wasserstein distance versus the computational time for all four competing algorithms. We observe from this figure that the ML2MC algorithm 4.1, clearly outperforms the competing algorithms. On an average (over the computational costs considered), it provides a speedup (reduction in error at same computational cost) of a factor of 5 over the Monte Carlo algorithm, 4 over the MLMC method and 3 over the single-level machine learning UQ algorithm, based on the deep learning algorithm 2.1.

	ML2MC Configurations				
Samples N_0	256	256	2048	2048	2048
Samples N_L	4	8	8	32	64
Complexity c_{ml}	0.17	0.67	0.67	0.67	1.5

Table 1: Different cost configurations used to evaluate the performance of multilevel machine learning method for the Projectile Motion example.

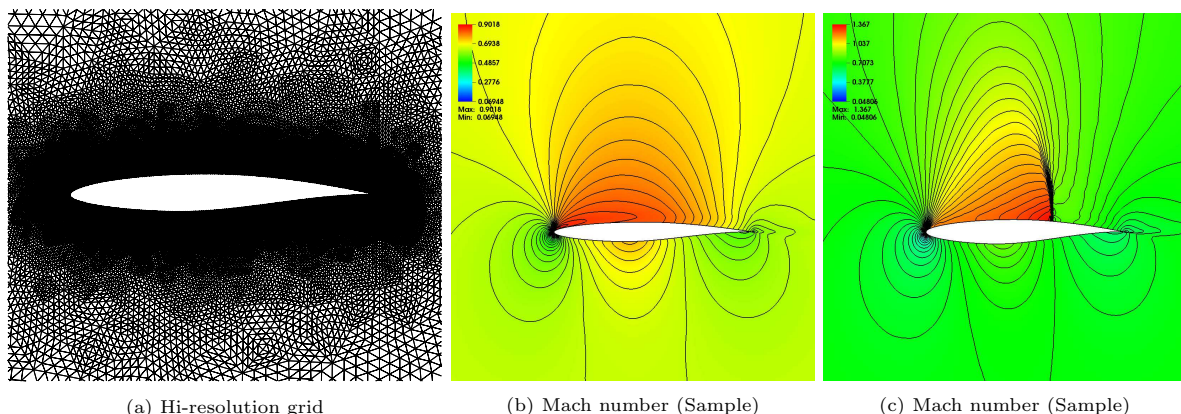(a) Hi-resolution grid   (b) Mach number (Sample)   (c) Mach number (Sample)

Figure 6: Flow past a RAE2822 airfoil. Left: High-resolution grid Center and Right: Flow visualized with Mach number for two different samples.

6.2 Flows past airfoil.

In this section, we consider a much more realistic example of a compressible flow past a RAE2822 airfoil. The problem set up is a benchmark for UQ in fluid flows [12] and is identical to the one considered in [19]. The two-dimensional compressible Euler equations are solved on the following perturbed free stream conditions and the profile geometry:

$$T^\infty(y) = (1 + \varepsilon_1 G_1(y)), \quad M^\infty(y) = 0.729(1 + \varepsilon_1 G_2(y)),$$

$$p^\infty(y) = (1 + \varepsilon_1 G_2(y)), \quad \alpha(y) = 2.31°(1 + \varepsilon_1 G_6(y)) \tag{6.6}$$

$$S_L(x; y) = \overline{S}_L(x)(1 + \varepsilon_2 G_4(y)), \quad S_U(x; y) = \overline{S}_U(x)(1 + \varepsilon_2 G_4(y)),$$

where α is the angle of attack and $\overline{S}_U(x)$, $\overline{S}_L(x)$, $x \in [0,1]$, denote the unperturbed upper and lower surfaces of the airfoil, respectively, see figure 6 (Left) for the reference geometry. G_k are defined as in the previous numerical examples and $\varepsilon_1 = 0.1$, $\varepsilon_2 = 0.2$. The observables are the lift and drag coefficients:

$$\mathcal{L}_1(y) = C_L(y) = \frac{1}{E_k^\infty(y)} \int_{S_L \cup S_U} p(y)n(y) \cdot \hat{y}(y)ds, \tag{6.7}$$

$$\mathcal{L}_2(y) = C_D(y) = \frac{1}{E_k^\infty(y)} \int_{S_L \cup S_U} p(y)n(y) \cdot \hat{x}(y)ds, \tag{6.8}$$

where $\hat{x}(y) = [\cos(\alpha(y)), \sin(\alpha(y))]$, $\hat{y}(y) = [-\sin(\alpha(y)), \cos(\alpha(y))]$ and

$$E_k^\infty(y) = \frac{\rho^\infty(y)||\mathbf{u}^\infty(y)||^2}{2}, \tag{6.9}$$

is the free-stream kinetic energy.

Thus, the input parameter space is the 6-dimensional cube $Y = [0,1]^6$ and two samples, corresponding to two different realizations in Y are shown in figure 6.

In [19], the authors observed that for this problem, the deep learning algorithm 2.1 with Sobol training points was significantly more accurate, with a factor of $10 - 20$ lower generalization errors than with randomly distributed training points. Hence, we will only consider the case of Sobol training points here. Moreover, we use this example to test the more general multi-level machine learning algorithm 5.1.

To this end, we consider 5 levels of grid resolution approximating the flow past the airfoil, with $L = 4$ in algorithm 5.1. The grid at the finest resolution is shown in figure 6 (Left). For each resolution, the two-dimensional Euler equations will be solved with the TEnSUM code, which implements vertex centered high-resolution finite volume schemes on unstructured triangular grids [31].

For implementing the multi-level algorithm, we consider 4 sequences of multi-level resolutions corresponding to $s_n^1 = \{0,4\}$ ($c_{ml} = 0.25$), $s_n^2 = \{0,2,4\}$ ($c_{ml} = 1.0$), $s_n^3 = \{0,2,3,4\}$ ($c_{ml} = 2.31$), $s_n^4 = \{0,1,2,3,4\}$ ($c_{ml} = 4.0$). Moreover, the same choices of samples at the coarsest level (N_0) and finest level (N_L) are made as in the previous numerical experiment. This results in a total of 112 multi-level hyperparameter configurations.

For the lift coefficient, we use the neural network architecture of [19], namely a fully-connected network of 9 hidden layers with 12 neurons in each layer, and minimize a mean square loss function with the ADAM optimizer at a fixed learning rate of $\eta = 0.01$. This configuration was used for learning both $\mathcal{L}_{0,NN}^*$ as well as the reference detail $\mathcal{D}_{ref,NN}^*$. An ensemble training, in the sense of [19], was performed to discover that a mean-square regularization with a $\lambda = 5 \times 10^{-7}$, provided the optimal hyperparameters for both maps. The same hyperparameter configuration was used for learning other details. Similarly for the Gaussian process regression hyperparameters, we performed an ensemble training on $\mathcal{L}_{0,GP}^*$, $\mathcal{D}_{ref,GP}^*$ to find that the Matern covariance kernel (5.3) with $\nu = 1.5$ provided the best hyperparameters.

For the drag coefficient, we retain the same hyperparameters for the Gaussian process, but for the neural networks in algorithm 5.1, we performed a full ensemble training for each map, which resulted in hyperparameters shown in table 2. In addition to those listed in this table, the ADAM optimizer was used to minimize a mean-square loss function.
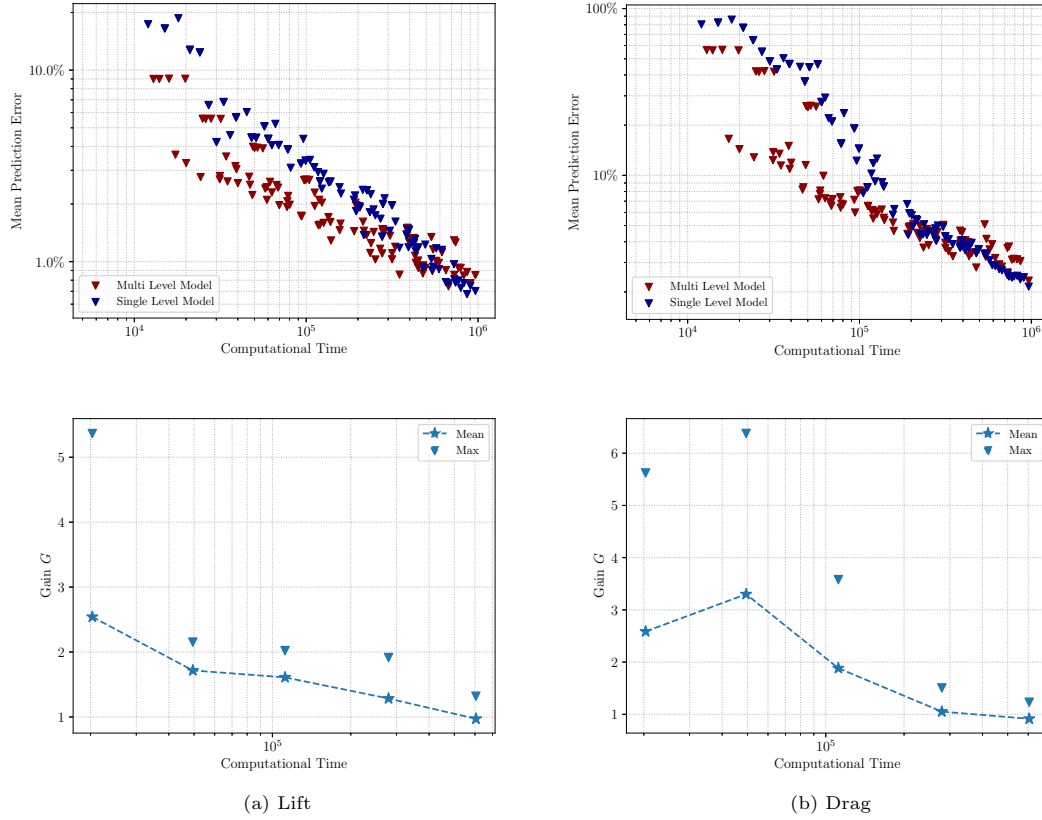
(a) Lift

(b) Drag

Figure 7: Prediction errors with the multi-level machine learning algorithm 5.1 for flow past a RAE2822 airfoil. Top Row: Prediction error vs. Computational cost. Bottom Row: Gain (6.5) over single level algorithm vs. computational cost.

	Samples	Lear. Rate	L^1 - Reg.	L^2 - Reg.	Depth	Width
$\mathcal{L}^{\Delta_0}$	1024	0.001	0	$5 \cdot 10^{-7}$	6	16
$\mathcal{L}^{\Delta_1} - \mathcal{L}^{\Delta_0}$	256	0.001	0	10^{-6}	6	12
$\mathcal{L}^{\Delta_2} - \mathcal{L}^{\Delta_1}$	64	0.001	$5 \cdot 10^{-7}$	0	6	12
$\mathcal{L}^{\Delta_3} - \mathcal{L}^{\Delta_2}$	64	0.01	$5 \cdot 10^{-6}$	0	6	8
$\mathcal{L}^{\Delta_4} - \mathcal{L}^{\Delta_3}$	8	0.01	10^{-5}	0	6	8
$\mathcal{L}^{\Delta_2} - \mathcal{L}^{\Delta_0}$	256	0.001	$5 \cdot 10^{-7}$	0	6	8
$\mathcal{L}^{\Delta_4} - \mathcal{L}^{\Delta_2}$	8	0.01	$5 \cdot 10^{-7}$	0	6	12
$\mathcal{L}^{\Delta_3} - \mathcal{L}^{\Delta_0}$	256	0.001	$5 \cdot 10^{-7}$	0	9	16
$\mathcal{L}^{\Delta_4} - \mathcal{L}^{\Delta_0}$	32	0.001	10^{-5}	0	6	8

Table 2: Best performing *Neural Network* configurations for the drag coefficient. In first column, we show the nnumber of training samples used in performing the ensemble training.

6.2.1 Prediction errors with multi-level model.

We approximate the generalization error (2.16) by computing the prediction error of the algorithms (6.4) on a *test set*, formed by 2000 Sobol training points on the finest mesh resolution. The resulting prediction errors for the multi-level machine learning algorithm 5.1, for all the 112 multi-level hyperparameters configurations, for both the lift and the drag, are shown in figure 7 (Top Row). To provide a comparison, we also compute the *single-level* surrogate map $\alpha_1 \mathcal{L}_{NN}^* + \alpha_2 \mathcal{L}_{GP}^*$, with all the training samples being generated at the finest grid resolution and determined to (approximately) match the computational cost of the multi-level algorithm and plot the corresponding prediction errors in figure 7. From this figure, we see that the multi-level algorithms clearly outperform the single-level machine learning algorithm, with approximately 75% of the configurations, resulting in a lower prediction error at similar computational cost. A few of the configurations (3% for the lift and 8% for the drag) result in at least a factor of 4 reduction in the prediction error at same cost.

This gain in efficiency is further quantified by computing the gain (6.5) and plotting it in figure 7 (Bottom Row). From this figure, we see that on an average, the gain with the multi-level model is a factor of $2-3$ for the Lift and $3-4$ for the Drag, in the range of reasonably small computational times, corresponding to the interesting case of low number of training samples. The maximum gain in this range is approximately 6 for both observables. Although these gains are a bit smaller than the ones for the projectile motion, they are more impressive as the underlying maps are hard to learn but the combination of Sobol training points, deep neural networks, and Gaussian processes makes the competing single-level machine learning model quite accurate.

6.2.2 Sensitivity of results to multi-level hyperparameters.

Next, we study the sensitivity of prediction errors and in particular, of the gains of the multi-level algorithm over the single-level algorithm, with respect to the three multi-level hyperparameters i.e, model complexity c_{ml}, number of training samples N_0 (at the coarsest level) and N_L (at the finest level). This sensitivity, for both the lift and the drag, is plotted in figure 8. We have the following observations from this figure,

- Sensitivity to N_0: We observe from figure 8 (Left column) that the best gains for the multi-level model arise when a larger number of samples is used at the coarsest level of resolution.

- Sensitivity to N_L: We observe from figure 8 (Middle column) that the best gains for the multi-level model arise when a few samples are used at the finest level. This is not surprising as increasing the number of samples at the finest level increases the overall cost dramatically, without possibly reducing the prediction error to the same extent.

- Sensitivity to c_{ml}: We observe from figure 8 (Right column) that the models of intermediate complexity, particularly with $c_{ml} = 1$, provide the best gains over the single-level model as they ensure a balance between accuracy and computational cost, when compared to the inaccurate models of low complexity and costly models of high complexity.

6.2.3 Uncertainty quantification

Finally, we consider uncertainty quantification (forward UQ) by computing approximations of the push-forward measure (probability distribution) (4.1), with respect to each observable. To this end, we use the generalized version of the ML2MC algorithm (by replacing the multi-level algorithm 3.1 with the extended multi-level algorithm 5.1). For comparison, a reference Quasi-Monte Carlo solution is computed with 2000 Sobol points.

We compute approximations to push-forward measure with multi-level configurations, shown in Table 3 for the lift and the drag. For the sake of comparison, we also compute the measure with the standard single-level Quasi-Monte Carlo algorithm (QMC), the multi-level Quasi-Monte Carlo (MLQMC) algorithm [20] and the single-level machine learning algorithm (SL2MC) trained with 4, 7, 16, 32, 64, 128, 144, 228, 256 samples for the lift and 4, 8, 16, 32, 110, 128 samples for the drag. For the MC method,

(a) Sensitivity to N_0 (b) Sensitivity to N_L (c) Sensitivity to c_{ml}
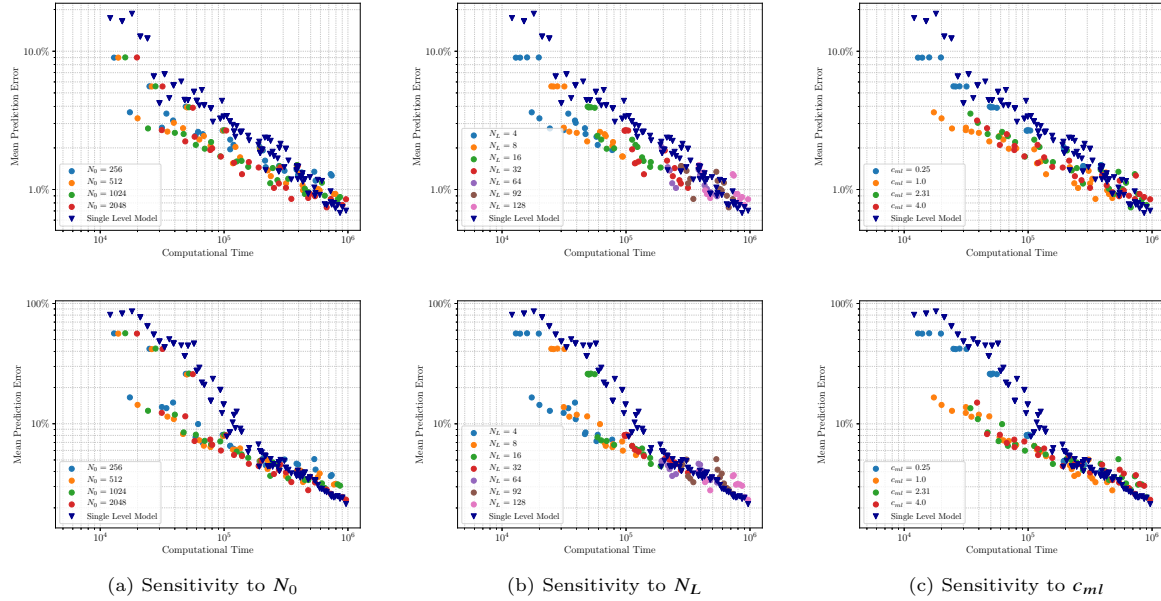
Figure 8: Sensitivity of the multi-level machine learning algorithm 5.1 to multi-level hyperparameters for the flow past RAE2822 airfoil. Top Row: Lift, Bottom Row: Drag. We plot prediction error (Y-axis) vs. Computational Cost (X-axis)
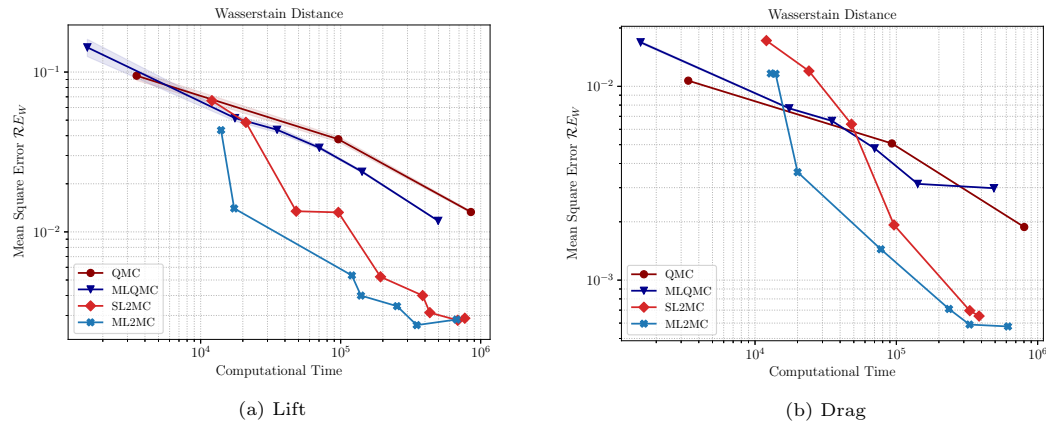

(a) Lift   (b) Drag

Figure 9: Uncertainty quantification for the RAE2822 airfoil. The Wasserstein distance between the approximate push-forward measure (4.1) computed with standard QMC, multi-level QMC (MLQMC), single-level machine learning (SL2MC) and multi-level machine learning (ML2MC) and a reference push-forward measure for the Lift and the Drag vs computational cost

23

	ML2MC Configurations							
Lift	**Samples** N_0	512	256	2048	2048	2048	2048	2048
	Samples N_L	4	4	8	32	64	92	92
	Complexity c_{ml}	0.25	1	4	1	1	1	2.31
Drag	**Samples** N_0	256	512	512	512	1024	1024	1024
	Samples N_L	4	4	4	8	64	92	92
	Complexity c_{ml}	0.25	0.25	1	4	1	1	2.31

Table 3: Different configurations for multilevel machine learning model for approximating the push forward measure wrt Lift and Drag.

we choose the number of samples such that the computational costs of the algorithms are comparable to the cost of the multi-level algorithm.

The results are shown in figure 9, where we plot the Wasserstein distance between the approximations and the reference measure, as a function of the computational cost. As seen from the figure, the UQ algorithm, based on the multi-level machine learning algorithm 5.1, clearly and substantially outperforms the standard QMC, multi-level QMC, and the single level machine learning algorithm for both the lift and the drag by providing considerably lower error for the same computational cost. In particular, we obtain an average speedup of 9.2 (over QMC), 5.4 (over multi-level QMC) and 1.7 (over single-level machine learning) for the lift and an average speedup of 4.8 (over QMC), 3.5 (over multi-level QMC) and 1.8 (over single-level machine learning) for the drag. The maximum speed-ups are even more impressive as they range from 11.4 (over QMC), 6.4 (over multi-level QMC) and 4.0 (over single-level machine learning) for the lift to an average speedup of 6.8 (over QMC), 5.2 (over multi-level QMC) and 3.8 (over single-level machine learning) for the drag. We would like to remark that the speed-ups can be even more substantial for the moments with respect to the push-forward measure. For instance for the standard deviation, we obtain speedup, on an average over the range of computational costs, of 5.9 (over QMC), 4.1 (over multi-level QMC) and 4.1 (over single-level machine learning) for the lift and an average speedup of 5.5 (over QMC), 3.3 (over multi-level QMC) and 3.0 (over single-level machine learning SL2MC) for the drag.

6.3 Code.

The assembling of the multilevel models and the ensemble training for the selection of the model hyperparameters are performed with a collection of Python scripts, with the support of Keras, Tensorflow and Scikit-learn. The scripts for the generation of the data set for the first numerical experiments and the construction of the multilevel model for both experiments can be downloaded from https://github.com/mroberto166/MultilevelMachineLearning.

7 Discussion

Machine learning, particularly deep learning, algorithms are increasingly popular in the context of scientific computing. One very promising area of application of these algorithms is in the computation of *observables*, corresponding to systems modeled by PDEs. The computation of these observables is very expensive as PDEs have to be simulated for every query i.e, every call of the underlying parameters to observable map (2.3). Instead, following [19], one can train deep neural networks to provide a surrogate for this parameters to observable map. Although it works well in practice, it was already observed in [19] that finding and training a neural network to approximate the parameters to observable map is very challenging as one can only expect to compute a few training data points (samples), given that the evaluation of each sample involves a very expensive PDE solve.

We tackle this issue in this paper and present a novel *Multi-level* algorithm to significantly increase the accuracy of deep learning algorithms, particularly in the poor data regime. The key idea behind our algorithm is based on the upper bound (2.19) on the generalization error of a deep learning algorithm for

regressing maps. We see from (2.19) that a significant component of this error is the variance (standard deviation) of the underlying map. Hence, variance reduction techniques can help reduce the generalization error.

Multi-level methods are examples for a class of variance reduction techniques, termed as control variate methods and are heavily used in the context of uncertainty quantification. We adapt the multi-level idea to machine learning algorithms. The main principle here is to simulate the training data on several mesh resolutions. A large number of cheap (computationally) training samples are used at coarse mesh resolutions whereas only a few computational expensive training samples at generated at fine mesh resolutions, to learn the details i.e differences between successive mesh resolutions.

We provide theoretical arguments in the form of lemmas 3.2 and 3.3 to support our contention that under some reasonable hypothesis on the training process, the multi-level machine-learning algorithm will lead to a decrease in generalization error, when compared to a single-level deep learning algorithm, at the same computational cost.

In fact, the same design principle works for a very general multi-level machine learning algorithm 5.1 that learns the parameters to observable map by a judicious combination of multi-level training on either random points or low-discrepancy sequences, deep neural networks and Gaussian process regressions.

We test the proposed algorithms on two representative problems. The first is a toy problem of ODEs modeling projectile motion but with a high-dimensional parameter space and highly non-linear parameters to observable map. The second problem is a benchmark uncertain compressible flow past an airfoil. For both problems, we observe that the multi-level algorithm significantly outperforms the single-level machine learning algorithm, resulting in computational gains from half an order to an order of magnitude, in the data poor regime. Moreover, we provide a recipe for finding the set of (multi-level) hyperparameters that result in the highest gains.

The multi-level algorithm is the basis for a machine-learning algorithm 4.1 for forward UQ or uncertainty propagation. Again we observe from the numerical experiments that the multi-level algorithm outperforms all competing algorithms that we tested and provided a computational gain of half an order to an order of magnitude over the standard MC (QMC), MLMC (MLQMC) and single-level machine learning models (such as the deep learning DLMC and DLQMC UQ algorithms of [19]).

Based on both theoretical and empirical results, we conclude that the multi-level algorithm provides a simple, straightforward to implement and efficient method for improving machine learning algorithms in the context of scientific computing.

Although we consider only the example of non-linear hyperbolic PDEs, the theory and the algorithms are readily extended to other PDEs such as elliptic and parabolic PDEs.

The results of this paper can be extended in many different directions, for instance the so-called *multi-fidelity* algorithms [26] can be readily adapted to machine learning and the multi-level algorithm can be extended to learn the whole solution field of the PDE (2.1), instead of to just the observable. Moreover, one can envisage applying multi-level techniques in more traditional applications of machine learning such as image and speech processing, where the multiple levels correspond to different resolutions of the image or the sound file. An interesting application of a related multi-level method in the context of computational chemistry is provided in the recent paper [37]. Finally, the multi-level machine learning algorithm 5.1 will be used in the context of shape optimization and Bayesian inverse problems in future work.

Acknowledgements.

References

[1] Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. *In Proceedings of the 35th International Conference on Machine Learning,* volume 80, pages 254-263. PMLR, Jul 2018.

[2] R. E. Caflisch. Monte Carlo and quasi-Monte Carlo methods. *Acta. Numer.*, 1, 1988, 1-49.

[3] G. Cybenko. Approximations by superpositions of sigmoidal functions. *Approximation theory and its applications.*, 9 (3), 1989, 17-28

[4] W. E and B. Yu. The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems. *Commun. Math. Stat.* 6 (1), 2018, 1-12.

[5] W. E, J. Han and A. Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Commun. Math. Stat.* 5 (4), 2017, 349-380.

[6] W. E, C. Ma and L. Wu. A priori estimates for the generalization error for two-layer neural networks. *ArXIV preprint*, available from arXiv:1810.06397, 2018.

[7] R. Evans et. al. De novo structure prediction with deep learning based scoring. *Google DeepMind working paper*, 2019.

[8] M. Giles. *Multilevel Monte Carlo path simulation.* Oper. Res., **56**:607-617, 2008.

[9] M. B. Giles. Multilevel Monte Carlo methods. *Acta Numer.,* 24 (2015), 259-328.

[10] I. Goodfellow, Y. Bengio, A. Courville. Deep learning. *MIT press, 2016*

[11] J. Han. A. Jentzen and W. E. Solving high-dimensional partial differential equations using deep learning. *PNAS*, 115 (34), 2018, 8505-8510.

[12] C. Hirsch, D. Wunsch, J. Szumbarksi, L. Laniewski-Wollk and J. pons-Prats (editors). *Uncertainty management for robust industrial design in aeronautics* Notes on numerical fluid mechanics and multidisciplinary design (140), Springer, 2018.

[13] S. Heinrich. *Multilevel Monte Carlo methods.* Large-scale scientific computing, Third international conference LSSC 2001, Sozopol, Bulgaria, 2001, Lecture Notes in Computer Science, Vol **2170**, Springer Verlag (2001), pp. 58-67.

[14] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks,* 2(5), 359-366, 1989.

[15] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, 1-13, 2015.

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

[17] Y. LeCun, Y. Bengio and G. Hinton. Deep learning. *Nature*, 521, 2015, 436-444.

[18] Sacks et al. Design and analysis of computer experiments. *Statistical science (1989)*: 409-423.

[19] K. O. Lye, S. Mishra, D. Ray. Deep learning observables in computational fluid dynamics. *arXiv preprint arXiv*:1903.03040 (2019).

[20] K. O. Lye. Statistical solutions of hyperbolic systems of conservation laws. *PhD Thesis,* ETH Zurich, In preparation.

[21] S. Mishra. A machine learning framework for data driven acceleration of computations of differential equations, *Math. in Engg.*, 1 (1), 2018, 118-146.

[22] S. Mishra and C. Schwab. Sparse tensor multi-level Monte Carlo finite volume methods for hyperbolic conservation laws with random initial data. *Math. Comput.*, 81(180), 2012, 1979–2018.

[23] S. Mishra, Ch. Schwab and J. Šukys. Multi-level Monte Carlo finite volume methods for nonlinear systems of conservation laws in multi-dimensions. *J. Comput. Phys* 231 (8), 2012, 3365–3388.

[24] T. P. Miyanawala and R. K, Jaiman. An efficient deep learning technique for the Navier-Stokes equations: application to unsteady wake flow dynamics. Preprint, 2017, available from arXiv :1710.09099v2.

[25] Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. Towards understanding the role of over-parametrization in generalization of neural networks. *arXiv preprint* arXiv:1805.12076, 2018.

[26] B. Peherstorfer, K. Willcox, and M. Gunzburger. Survey of multifidelity methods in uncertainty propagation, inference, and optimization. *SIAM Rev.* 60 (2018), no. 3, 550591.

[27] M. Raissi and G. E. Karniadakis. Hidden physics models: machine learning of nonlinear partial differential equations. *J. Comput. Phys.,* 357, 2018, 125-141.

[28] M. Raissi, A. Yazdani and G. E. Karniadakis. Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for assimilating flow visualization data. *ArXIV preprint*, available from arXiv:1808.04327, 2018.

[29] C.E Rasmussen. Gaussian processes in machine learning. *Summer School on Machine Learning*, Springer, Berlin, Heidelberg, 2003.

[30] D. Ray and J. S, Hesthaven. An artificial neural network as a troubled cell indicator. *J. Comput. Phys.,* 367, 2018, 166-191

[31] D. Ray, P. Chandrasekhar, U. S. Fjordholm and S. Mishra. Entropy stable scheme on two-dimensional unstructured grids for Euler equations, *Commun. Comput. Phys.,* 19 (5), 2016, 1111-1140.

[32] S. Ruder. An overview of gradient descent optimization algorithms. Preprint, 2017, available from arXiv.1609.04747v2.

[33] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[34] J. Tompson, K. Schlachter, P. Sprechmann and K. Perlin. Accelarating Eulerian fluid simulation with convolutional networks. *Preprint*, 2017. Available from arXiv:1607.03597v6.

[35] A. Quateroni, A. Manzoni and F. Negri. *Reduced basis methods for partial differential equations: an introduction*, Springer Verlag (2015).

[36] D. Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 94, 2017, 103-114

[37] P. Zaspel, B. Huang, H. Harbrecht and O. Anotole von Lillenfeld. Boosting quantum machine learning with multi-level combination technique: Pople diagrams revisited. Preprint, available as Arxiv1808.02799v2.

[38] Statistical functions (scipy.stats). Python Library. `https://docs.scipy.org/doc/scipy/reference/stats.html`