ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Seminar for Applied Mathematics SAM

Deep learning observables in computational fluid dynamics

K. Lye and S. Mishra and D. Ray

Research Report No. 2019-13

March 2019
Latest revision: March 2019

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

Funding ERC: 770880 COMANFLO

Deep learning observables in computational fluid dynamics

Kjetil O. Lye [*], Siddhartha Mishra [†] and Deep Ray [‡]

March 7, 2019

Abstract

Many large scale problems in computational fluid dynamics such as uncertainty quantification, Bayesian inversion, data assimilation and PDE constrained optimization are considered very challenging computationally as they require a large number of expensive (forward) numerical solutions of the corresponding PDEs. We propose a machine learning algorithm, based on deep artificial neural networks, that learns the underlying *input parameters to observable* map from a few training samples (computed realizations of this map). By a judicious combination of theoretical arguments and empirical observations, we find suitable network architectures and training hyperparameters that result in robust and efficient neural network approximations of the parameters to observable map. Numerical experiments for realistic high dimensional test problems, demonstrate that even with approximately 100 training samples, the resulting neural networks have a prediction error of less than one to two percent, at a computational cost which is several orders of magnitude lower than the cost of the underlying PDE solver.

Moreover, we combine the proposed deep learning algorithm with Monte Carlo (MC) and Quasi-Monte Carlo (QMC) methods to efficiently compute uncertainty propagation for nonlinear PDEs. Under the assumption that the underlying neural networks generalize well, we prove that the deep learning MC and QMC algorithms are guaranteed to be faster than the baseline (quasi-) Monte Carlo methods. Numerical experiments demonstrating one to two orders of magnitude speed up over baseline QMC and MC algorithms, for the intricate problem of computing probability distributions of the observable, are also presented.

1 Introduction

Many interesting fluid flows are modeled by so-called *convection-diffusion* equations [27] i.e, nonlinear partial differential equations (PDEs) of the generic form,

$$\mathbf{U}_t + \mathrm{div}_x(\mathbf{F}(\mathbf{U})) = \nu \ \mathrm{div}_x(\mathbf{D}(\mathbf{U})\nabla_x \mathbf{U}), \quad (x,t) \in D \subset \mathbb{R}^{d_s} \times \mathbb{R}_+, \tag{1.1}$$

with $\mathbf{U} \in \mathbb{R}^m$ denoting the vector of unknowns, $\mathbf{F} = (F_i)_{1 \leqslant i \leqslant d_s}$ the *flux vector*, $\mathbf{D} = (D_{ij})_{1 \leqslant i,j \leqslant d_s}$ the *diffusion matrix* and ν a small scale parameter representing kinematic viscosity.

Prototypical examples for (1.1) include the compressible Euler equations of gas dynamics, shallow water equations of oceanography and the magnetohydrodynamics (MHD) equations of plasma physics. These PDEs are *hyperbolic systems of conservation laws* i.e, special cases of (1.1) with $\nu = 0$ and with the flux Jacobian $\partial_{\mathbf{U}}(\mathbf{F} \cdot n)$ having real eigenvalues for all normal vectors n. Another important example for (1.1) is provided by the incompressible Navier-Stokes equations, where $0 < \nu << 1$ and the flux function is *non-local* on account of the divergence-free constraint on the velocity field.

It is well known that solutions to convection-diffusion equations (1.1) can be very complicated. These solutions might include singularities such as shock waves and contact discontinuities in the case of hyperbolic systems of conservation laws. For small values of ν, these solutions can be generically unstable, even turbulent, and contain structures with a large range of spatio-temporal scales [27].

[*]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
[‡]Institute of Mathematics, EPFL, Lausanne -1015, Switzerland.

Numerical schemes play a key role in the study of fluid flows and a large variety of robust and efficient numerical methods have been designed to approximate them. These include (conservative) finite difference [29], finite volume [19, 22], discontinuous Galerkin (DG) finite element [22] and spectral (viscosity) methods [47]. These methods have been extremely successful in practice and are widely used in science and engineering today.

The exponential increase in computational power in the last decades provides us with the opportunity to solve very challenging large scale problems in computational fluid dynamics, such as uncertainty quantification (UQ) [4, 17], (Bayesian) inverse problems [45] and real-time optimal control, design and PDE constrained (shape) optimization [6, 48]. In such problems, one is not always interested in computing the whole solution field $\mathbf{U}$ of (1.1). Rather and in analogy with experimental measurements, one is interested in computing the so-called *observables* (functionals or quantities of interest) for the solution $\mathbf{U}$ of (1.1). These can be expressed in the generic form,

$$L(\mathbf{U}) = \int\limits_{D\times\mathbb{R}_+} \psi(x,t) g(\mathbf{U}(x,t)) dx dt. \tag{1.2}$$

Here $\psi : D\times\mathbb{R}_+ \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}$ are suitable test functions. Prototypical examples of such observables (functionals) are provided by body forces, such as the *lift* and *drag* in aerodynamic simulations, and by the runup height (at certain probe points) in simulations of tsunamis.

Moreover in practice, one is interested not just in a single value, but rather in the *statistics* of such observables. Typical statistical quantities of interest are the mean, variance, higher-moments and probability density functions (pdfs) of the observable (1.2). Such statistical quantities quantify uncertainty in the solution, propagated from possible uncertainty in the underlying inputs i.e, fluxes, diffusion coefficients, initial and boundary data. They might also stem from the presence of a statistical spread in design parameters such as those describing the geometry of the computational domain. It is customary to represent the resulting solution field as $\mathbf{U} = \mathbf{U}(y)$, with $y \in Y \subset \mathbb{R}^d$, a possibly very high dimensional parameter space. Thus, the goal of many CFD simulations is to compute (statistics of) the so-called *parameters to observable* map $y \mapsto L(\mathbf{U}(y))$.

Calculating a single realization of this parameters to observable map might require a very expensive forward solve of (1.1) with a CFD solver and a quadrature to calculate (1.2). However, in UQ, Bayesian inversion or optimal design and control, one needs to evaluate a large number of instances of this map, necessitating a very high computational cost, even on state of the art HPC systems. As a concrete example, we consider a rather simple yet prototypical situation. We are interested in computing statistics of body forces such as lift and drag for a model two-dimensional RAE2822 airfoil [23] (see figure 17). A single forward solve of the underlying compressible Euler equations on this airfoil geometry on a mesh of high resolution (see figure 18), with a state of the art high-resolution finite volume solver [40], takes approximately 7 wall-clock hours on a HPC cluster (see table 13). This cost scales up in three space dimensions proportionately. However, a typical UQ problem such as Bayesian inversion with a state of the art MCMC algorithm, for instance the Random walk Metropolis-Hastings algorithm [45], might need upto $10^5 - 10^6$ such realizations, rendering even a two-dimensional Bayesian inversion infeasible ! This examples illustrates the fact that the high-computational cost of the parameters to observable map makes problems such as forward UQ, Bayesian inversion, and optimal control and design very challenging.

Although approaches such as model order reduction [49] have been developed to provide faster surrogates for the parameters to observable map, it is well known that such surrogates or reduced models are not stable or efficient enough for complex problems with shocks and/or turbulence, to be much practical utility. Hence, *we clearly need alternative methods which allow ultrafast (several order of magnitude faster than state of the art) computations of the underlying parameters to observable map.*

Machine learning, in the form of artificial neural networks (ANNs), has become extremely popular in computer science in recent years. This term is applied to methods that aim to approximate functions with layers of units (neurons), connected by (affine) linear operations between units and nonlinear activations within units, [18] and references therein. *Deep learning*, i.e an artificial neural network with a large number of intermediate (hidden) layers has proven extremely successful at diverse tasks, for instance in image processing, computer vision, text and speech recognition, game intelligence and more recently in protein folding [14], see [28] and references therein for more applications of deep learning. A key element in deep learning is the *training* of tunable parameters in the underlying neural network by

(approximately) minimizing suitable *loss functions*. The resulting (non-convex) optimization problem, on a very high dimensional underlying space, is customarily solved with variants of the stochastic gradient descent method [42].

Deep learning is being increasingly used in the context of numerical solution of partial differential equations. Given that neural networks are very powerful universal function approximators [9, 24, 3, 33, 51], it is natural to consider the space of neural networks as an ansatz space for approximating solutions of PDEs. First proposed in [26] on an underlying collocation approach, it has been successfully used recently in different contexts in [37, 38, 32, 11, 12, 21] and references therein. Given spatial and temporal locations as inputs, the (deep) neural networks, proposed by these authors, approximate the solution of the underlying PDE, by outputting function values. This approach appears to work quite well for problems with high regularity (smoothness) of the underlying solutions (see [43, 21]) and/or if the solution of the underlying PDE possesses a representation formula in terms of integrals [12, 21]. However, solutions of PDEs modeling fluid flows such as (1.1) are not quite regular, due to the presence of shocks or sharp gradients, nor can they be represented in terms of integral solution formulas. Hence it is unclear if complicated solutions of (1.1), realized as functions of space and time, can be efficiently learned by deep neural networks [46].

Several papers applying deep learning techniques in the context of CFD advocate embedding deep learning modules within existing CFD codes to increase their efficiency. As examples, one can consider solving the elliptic equations in a divergence projection step in incompressible flows [46] or learning troubled cell indicators within an RKDG code for applying limiters [39] or recasting finite difference (volume) schemes as neural networks and training the underlying parameters to improve accuracy on coarse grids [31].

In this paper, we adopt a different approach and propose to use *deep neural networks* to *learn the input parameters to observable map*. Our algorithm will be based on fully connected networks (multi-layer preceptrons) which output values of the observable (1.2) for different input parameters $y \in Y \subset \mathbb{R}^d$ with $d >> 1$. The network will be trained on data, generated from a few, say $O(100)$, *samples* i.e, realizations of (1.2) with expensive CFD solvers. Suitable loss functions need to be defined and minimized with a stochastic gradient descent method.

However, the task of designing deep neural networks that will approximate the parameters to observable map with reasonable accuracy is far from straightforward on account of the following issues,

- All available approximation results for deep neural networks stipulate that approximating a function to an error of size ε in any L^p norm, requires a network of size (number of free parameters) $O\left(\varepsilon^{\frac{-d}{s}}\right)$, with s denoting the Sobolev regularity of the underlying map. In other words, these estimates require bounds on the s-th derivative of the underlying function, see [51, 5, 36] and references therein. However, such bounds are too prohibitive for typical parameters to observable maps encountered in CFD. As explained in section 2.3, it is unreasonable to expect bounds on derivatives of this map beyond the first derivative, on account of the presence of shocks and/or turbulence. Even first derivative bounds are not necessarily available. Thus in this context, approximation theory suggests networks of unrealistically large sizes even for problems with moderate dimensional input parameter spaces .

- One can rightly argue that approximation theory bounds for neural networks are not relevant for us as we *train* our networks, with the stochastic gradient method, on available data. Rather, the relevant quantitative measure is the so-called *generalization error* (see section 2.3), that measures the error with trained networks on *unseen data*. The best available bounds on generalization error scale as $O\left(\sqrt{\frac{U}{M}}\right)$, with M being the number of training samples. The numerator U is usually bounded above by the so-called Vapnik-Chervonenkis (VC) dimension or Rademacher complexity [44]. Unfortunately, these bounds are rather pessimistic in practice and grossly overestimate the generalization error, [2] and references therein. In fact, the number of tunable parameters in the neural network is considered to be a more tight upper bound for U than these measures [2]. Even if we assume that $U \sim O(1)$, to achieve an accuracy of one percent relative error, requires $O\left(10^4\right)$ samples. This is way more than the $O(100)$ high resolution samples that we can compute in practice,

at least for three space dimensions. Thus, we are in a relatively *data poor* regime in CFD, which is in stark contrast to the big data applications that machine learning excels in.

Hence, *it is quite challenging to find deep neural networks that can accurately approximate maps of low Sobolev regularity in a data poor regime.* Moreover, there are several hyperparameters that need to be specified in this framework, for instance size of the networks (number of layers, width of each layer), choice of which variant of the stochastic gradient algorithm that one uses, choice of loss functions and regularizations thereof, which data points to sample from, etc. A priori, results with neural networks can be sensitive to these choices. We propose an *ensemble training* procedure to search the hyperparameter space systematically and identify network architectures that are efficient in our context. By a combination of theoretical considerations and rigorous empirical experimentation, we provide a recipe for finding appropriate deep neural networks to compute the parameters to observable map i.e, those network architectures which ensure a low generalization error, even for relatively few training samples.

A second aim of this paper is to apply these trained deep neural networks in context of uncertainty propagation (forward UQ). We will combine the proposed deep neural networks together with Monte Carlo and Quasi-Monte Carlo statistical sampling procedures to compute statistical quantities of interest for observables in CFD. In particular, we will be interested in computing probability distributions (measures) of the observable and demonstrate that combining deep learning with (quasi)-Monte Carlo is significantly more efficient than baseline algorithms.

The rest of the paper is organized as follows: the deep learning algorithm is presented in section 2. The deep learning (Quasi-)Monte Carlo algorithm for forward UQ is presented in section 3 and some details for the implementation of both sets of algorithms are provided in section 4. In section 5, we present numerical experiments illustrating the proposed algorithms and the results of the paper are summarized and discussed in section 6.

2 Deep learning algorithm

2.1 The problem

We consider the following very general form of a *parameterized* convection-diffusion equation,

$$\begin{aligned} \partial_t \mathbf{U}(t,x,y) + \mathrm{div}_x(\mathbf{F}(y,\mathbf{U})) &= \nu \ \mathrm{div}_x(\mathbf{D}(y,\mathbf{U})\nabla_x \mathbf{U}), \quad \forall\ (t,x,y) \in [0,T] \times D(y) \times Y, \\ \mathbf{U}(0,x,y) &= \overline{\mathbf{U}}(x,y), \quad \forall\ (x,y) \in D(y) \times Y, \\ L_b \mathbf{U}(t,x,y) &= \mathbf{U}_b(t,x,y), \quad \forall\ (t,x,y) \in [0,T] \times D(y) \times Y \end{aligned} \tag{2.1}$$

Here, Y is the 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 flux vector is denoted as $\mathbf{F} = (F_i)_{1 \leqslant i \leqslant d_s} : \mathbb{R}^m \times Y \to \mathbb{R}^m$ and $\mathbf{D} = (D_{ij})_{1 \leqslant i,j \leqslant d_s} : \mathbb{R}^m \to \mathbb{R}^m$ is the *diffusion matrix*.

The operator L_b is a *boundary* operator that imposes boundary conditions for the PDE, for instance the no-slip boundary condition for incompressible Navier-Stokes equations or characteristic boundary conditions for hyperbolic systems of conservation laws. Additional conditions such as hyperbolicity for the flux function $\mathbf{F}$ and positive-definiteness of the diffusion matrix $\mathbf{D}$ might also be imposed, depending on the specific problem that is being considered.

We remark that the parameterized PDE (2.1) will arise in the context of (both forward and inverse) UQ, when the underlying convection-diffusion equation (1.1) contains uncertainties in the domain, the flux and diffusion coefficients and in the initial and boundary data. This input uncertainty propagates into the solution. Following [4] and references therein, it is customary to model such random inputs and the resulting solution uncertainties by *random fields*. Consequently, one can parameterize the probability space on which the random field is defined in terms of a parameter space $Y \subset \mathbb{R}^d$, for instance by expressing random fields in terms of (truncated) Karhunen-Loeve expansions. By normalizing the resulting random variables, one may assume $Y = [0,1]^d$, with possibly a large value of the parameter dimension d. Moreover, there exists a measure, $\mu \in \mathrm{Prob}(Y)$, with respect to which the data from the underlying parameter space is drawn.

We point out that the above framework is also relevant in problems of optimal control, design and PDE constrained optimization. In these problems, the parameter space Y represents the set of design or control parameters.

For the parameterized PDE (2.1), we aim to compute observables of the following general form,

$$L_g(y, \mathbf{U}) := \int_0^T \int_{D_y} \psi(x,t) g(\mathbf{U}(t,x,y)) dx dt, \quad \text{for } \mu \text{ a.e } y \in Y. \tag{2.2}$$

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 $s \geqslant 1$. Most interesting observables encountered in experiments, such as the lift and the drag, can be cast in this general form.

For fixed functions ψ, g, we also 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 convection-diffusion equation (2.1) for every parameter vector $y \in Y$. These schemes could be of the finite difference, finite volume, DG or spectral type, depending on the problem and on the baseline CFD code. Hence for any mesh parameter (grid size, time step) Δ, we are assuming that for any parameter vector $y \in Y$, a high-resolution approximate solution $\mathbf{U}^\Delta(y) \approx \mathbf{U}(y)$ is available. Hence, there exists an approximation to the *input to observable* map $\mathcal{L}$ of the form,

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

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 $\Delta << 1$, 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,

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

for $1 \leqslant p < \infty$. The L^∞_μ norm is analogously defined.

2.2 Deep learning the parameters to observable map

As stated earlier, it can be very expensive to compute the map $\mathcal{L}^\Delta(y)$ for each single realization, $y \in Y$, as a high-resolution CFD solver, possibly entailing a very large number of degrees of freedom, needs to be used. We propose instead, to *learn* this map by deep neural networks. This process entails the following steps,

2.2.1 Training set.

As is customary in supervised learning [18] 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 as,

(i.) randomly from Y, independently and identically distributed with the underlying probability distribution μ.

(ii.) from a suitable set of quadrature points in Y, for instance the so-called *low discrepancy sequences* that arise in Quasi-Monte Carlo (QMC) quadrature algorithms [8]. Examples of such sequences include Sobol or Halton QMC quadrature points [8].

We emphasize that the generation of QMC points is very cheap, particularly for Sobol or Halton sequences. Moreover, these points are better spread out over the parameter space than a random selection of points and might provide more detailed information about it [8]. Hence, a priori QMC points might be a better choice for sampling the data. One can also replace QMC points with other hierarchical algorithms such as nodes of sparse grids (Smolyak quadrature points) [7] to form the set $\mathcal{S}$.

Once the training parameter set $\mathcal{S}$ is chosen, we perform a set of high-resolution CFD simulations to obtain $\mathcal{L}^{\Delta}(y)$, for all $y \in \mathcal{S}$. As each high-resolution CFD simulation could be very expensive, we will require that $N = \#(\mathcal{S})$ will not be very large. It will typically be of at most $\mathcal{O}(100)$.

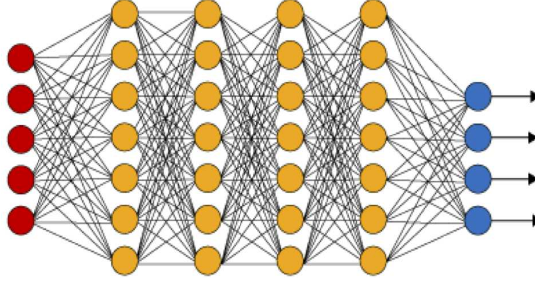


Figure 1: An illustration of a (fully connected) deep neural network. The red neurons represent the inputs to the network and the blue neurons denote the output layer. They are connected by hidden layers with yellow neurons. Each hidden unit (neuron) is connected by affine linear maps between units in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

2.2.2 Neural network.

Given an input vector, a feedforward neural network (also termed as a multi-layer perceptron), shown in figure 1, consists of layer of units (neurons) which compose of either affine-linear maps between units (in successive layers) or scalar non-linear activation functions within units, culminating in an output [18]. In our framework, for any input vector $z \in Y$, we represent an artificial neural network as,

$$\mathcal{L}_{\theta}(z) = C_K \odot \sigma \odot C_{K-1} \ldots\ldots\ldots \odot \sigma \odot C_2 \odot \sigma \odot C_1(z). \tag{2.6}$$

Here, $\odot$ 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 [18]. A very popular choice is the *ReLU* function,

$$\sigma(z) = \max(z, 0). \tag{2.7}$$

When, $z \in R^p$ for some $p > 1$, then the output of the ReLU function in (2.7) is evaluated componentwise.

Moreover, for any $1 \leqslant k \leqslant K$, we define

$$C_k z_k = W_k z_k + b_k, \quad \text{for } W_k \in \mathbb{R}^{d_{k+1} \times d_k}, z_k \in \mathbb{R}_{d_k}, b_k \in \mathbb{R}^{d_{k+1}}. \tag{2.8}$$

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 (2.6) 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 (2.8) and then by a ReLU (or another) nonlinear (component wise) activation σ (2.7). Although the neural network consists of composition of very elementary functions, its complexity and ability to learn very general functions arises from the interactions between large number of hidden layers [18].

A straightforward addition shows that our network contains $\left(d + 1 + \sum\limits_{k=2}^{K-1} d_k\right)$ neurons.

We denote,

$$\theta = \{W_k, b_k\}, \theta_W = \{W_k\} \quad \forall\ 1 \leqslant k \leqslant K, \tag{2.9}$$

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

$$M = \sum_{k=1}^{K-1} (d_k + 1) d_{k+1}. \tag{2.10}$$

Thus, depending on the dimensions of the input parameter vector and the number (depth) and size (width) of the hidden layers, our proposed neural network can contain a large number of weights. Moreover, the neural network explicitly depends on the choice of the weight vector $\theta \in \Theta$, justifying the notation in (2.6).

Although a variety of network architectures, such as convolutional neural networks or recurrent neural networks, have been proposed in the machine learning literature, [18] and references therein, we will restrict ourselves to fully connected architectures i.e, we do not a priori assume any *sparsity* structure for our set Θ.

2.2.3 Loss functions and optimization.

For any $y \in \mathcal{S}$, we have already evaluated $\mathcal{L}^\Delta(y)$ from the high-resolution CFD simulation. 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 *loss function* or mismatch function, as

$$J(\theta) := \sum_{y \in \mathcal{S}} |\mathcal{L}^\Delta(y) - \mathcal{L}_\theta(y)|^p, \tag{2.11}$$

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 (2.11) is minimized. The resulting optimization (minimization) problem might lead to searching a minimum of a non-convex loss function. So, it is not uncommon in machine learning [18] to regularize the minimization problem i.e we seek to find,

$$\theta^* = \arg\min_{\theta \in \Theta} \left(J(\theta) + \lambda \mathcal{R}(\theta) \right). \tag{2.12}$$

Here, $\mathcal{R} : \Theta \mapsto \mathbb{R}$ is a *regularization* (penalization) term. A popular choice is to set $\mathbb{R}(\theta) = \|\theta_W\|_q^q$ for either $q = 1$ (to induce sparsity) or $q = 2$. The parameter $0 < \lambda << 1$ balances the regularization term with actual loss J (2.11).

The above minimization problem amounts to finding a minimum of a possibly non-convex function over a subset of $\mathbb{R}^M$ for very large M. We can approximate the solutions to this minimization problem iteratively, either by a full batch gradient descent algorithm or by a mini-batch stochastic gradient descent (SGD) algorithm. A variety of SGD algorithms have been proposed in the literature and are heavily used in machine learning, see [42] for a survey. A generic step in a (stochastic) gradient method is of the form:

$$\theta_{r+1} = \theta_r - \eta_r \nabla_\theta \left(J(\theta_k) + \lambda \mathcal{R}(\theta_k) \right), \tag{2.13}$$

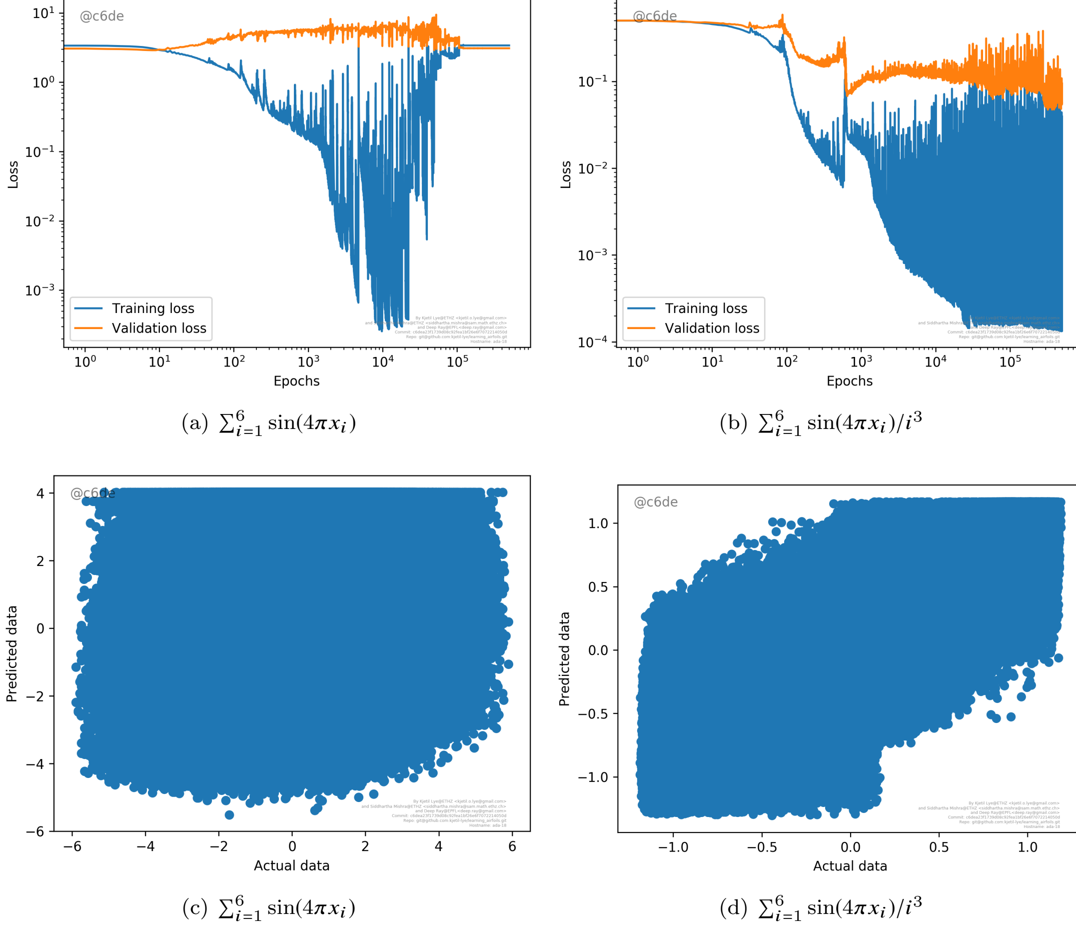
with η_r being the *learning rate*. The stochasticity arises in approximating the gradient in (2.13) by,

$$\nabla_\theta J(\theta_k) \approx \nabla_\theta \left(\sum_{y \in \hat{\mathcal{S}}_q} |\mathcal{L}^\Delta(y) - \mathcal{L}_\theta(y)|^p \right), \tag{2.14}$$

and analogously for the gradient of the regularization term in (2.13). Here $\hat{\mathcal{S}}_q \subset \mathcal{S}$ refers to the q-th batch, with the batches being shuffled randomly. Moreover, the SGD methods are initialized with a starting value $\theta_0 = \bar{\theta} \in \Theta$. A widely used variant of the SGD method is the so-called *ADAM* algorithm [25].

For notational simplicity, we denote the (approximate, local) minimum weight vector in (2.12) as θ^* and the underlying deep neural network $\mathcal{L}_{\theta^*}$ 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.



(a) $\sum_{i=1}^{6} \sin(4\pi x_i)$ (b) $\sum_{i=1}^{6} \sin(4\pi x_i)/i^3$

(c) $\sum_{i=1}^{6} \sin(4\pi x_i)$ (d) $\sum_{i=1}^{6} \sin(4\pi x_i)/i^3$

Figure 2: Results with neural networks, with architecture tabulated in table 3 and hyperparameters specified in table 5, approximating the (scaled) sum of sines (2.29) in six dimensions. Top row: Training (Validation) loss with respect to number of training epochs. Bottom Row: Scatter plot with ground truth (x-axis) vs predicted values of trained neural networks (y-axis). Errors are proportional to the spread away from the diagonal.

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 high-resolution CFD simulations.*

Step 2: *For an initial value of the weight vector* $\overline{\theta} \in \Theta$, *evaluate the neural network* $\mathcal{L}_{\overline{\theta}}$ (2.6), *the loss function* (2.12) *and its gradients to initialize the (stochastic) gradient descent algorithm.*

Step 3: *Run a stochastic gradient descent algorithm of form* (2.13) *till an approximate local minimum* θ^* *of* (2.12) *is reached. The map* $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ *is the desired neural network approximating the parameters to observable map* $\mathcal{L}$.

2.3 Theory

2.3.1 Approximation with deep neural networks.

It is well known that neural networks can approximate very general functions. Some of the early *universal approximation theorems* for neural networks [3, 24, 9] showed that a shallow two-layer network suffices to approximate any continuous (in fact any measurable) function. However, no explicit quantitative estimates are provided on the number of units in the hidden layer. Spurred by the tremendous success achieved by neural networks in computer science, a large number of much sharper approximation results are now available . We state one the of the benchmark approximation results below,

Theorem 2.2. *Let $f \in W^{s,p}\left([0,1]^d, \mathbb{R}\right)$ for some $1 \leqslant p \leqslant \infty$, such that $\|f\|_{W^{s,p}} \leqslant 1$, then for every $\varepsilon > 0$, there exists a neural network $NN(f)$ of the form (2.6) with the ReLU activation function, with $O\left(1 + \log\left(\frac{1}{\varepsilon}\right)\right)$ layers and of size (number of weights) $O\left(\varepsilon^{-\frac{d}{s}}\left(1 + \log\left(\frac{1}{\varepsilon}\right)\right)\right)$ such that*

$$\|f - NN(f)\|_{L^p} \leqslant \varepsilon, \quad 1 \leqslant p \leqslant \infty. \tag{2.15}$$

This theorem is essentially a restatement of Theorem 3.2 of [51]. However, in [51], only the $p = \infty$ case was considered. The general p case was considered in a recent paper [36]. It should be mentioned that one can replace the $W^{s,p}$ assumption by a *piecewise-C^s* assumption as in corollary 3.7 of [36]. The proof of both results relies on the efficient approximation of multiplication of two scalars by neural networks of small size and using the approximability of smooth functions by polynomials. Moreover, certain lower bounds have been derived in [51, 36] and references therein, which show that the above estimate on the network size (number of weights) is essentially optimal if one considers general (piecewise) smooth functions.

We remark that theorem 2.2 claims such an optimal neural network exists but does not provide any constructive algorithm to find this network. However, the estimates on network size might provide an useful basis for deciding the size and architecture of the network (2.6), that will approximate the parameters to observable map.

2.3.2 Regularity of the parameters to observable map (2.3)

In order to apply an approximation theorem such as theorem 2.2 in our setting, we need to determine the regularity of the underlying parameters to observable map $\mathcal{L}$ (2.3) or equivalently its numerical approximation $\mathcal{L}^\Delta$ (2.4).

For simplicity, we assume that the underlying domain $D_y = D \subset B_R$ for a. e. $y \in Y$ and B_R is a ball of radius R. Moreover, we assume that $\psi, g^k \in L^\infty$, with g^k denoting the k-th derivative of the map g in (2.2), for some $k \geqslant 1$. Then, it is straightforward to obtain the following upper bound,

$$\|\mathcal{L}\|_{W^{k,p}(Y)} \leqslant C\|\mathbf{U}\|_{W^{k,p}(D\times(0,T)\times Y)}, \tag{2.16}$$

with the constant C depending only on the initial data, Ψ and g. Thus, regularity in the parameters to observable map is predicated on the space-time and parametric regularity of the underlying solution field $\mathbf{U}$.

Do solutions of the convection-diffusion PDE (1.1) possess such Sobolev regularity ? It is instructive to start with a very simple example – that of the one-dimensional *parameterized* Burgers' equation:

$$\begin{aligned} \partial_t u(t,x,y) + \partial_x\left(\frac{(u(t,x,y)^2}{2}\right) = 0, \quad x \in D, \quad t \in (0,T), \quad \forall y \in Y \\ u(0,x,y) = u_0(x,y), \quad x \in D, \quad \forall y \in Y. \end{aligned} \tag{2.17}$$

Here, $u = u(t,x,y) : (0,T) \times D \times Y \mapsto \mathbb{R}$ is the sought after solution field. For the purpose of illustration, we assume that $D = \mathbb{R}$ and $Y = [0,1]$.

In this simple case, the regularity of the map $\mathcal{L}$ (2.3) is readily estimated in terms of the integrability of u_y, u_{yy} and higher parametric derivatives of u. Can we estimates these parametric derivatives ? The following straightforward formal calculation illustrates this process for the first parametric derivative u_y.

We start by setting $w = u_y$ and (formally) differentiating (2.17) with respect to the parameter y resulting in,

$$w_t + uw_x + u_x w = 0, \quad \forall y \in Y. \tag{2.18}$$

Now multiplying the above equation (2.18) with $p\text{sign}(w)|w|^{p-1}$ (for $1 < p < \infty$) yields,

$$(|w|^p)_t + u\,(|w|^p)_x + p|w|^p u_x = 0.$$

Integrating the above over space (assuming that the function u decays at infinity) and integrating by parts results in,

$$\frac{d}{dt}\int_D |w|^p dx = -(p-1)\int_D u_x |w|^p dx. \tag{2.19}$$

The above equation indicates the difficulty of estimating the parametric derivative $w = u_y$ in $L^p(Y)$ for $1 < p \leqslant \infty$. As long as $u \in W^{1,\infty}(D \times (0,T) \times Y)$, we can use Grönwall inequality on the above equation to conclude that

$$\|u_y\|_{L^\infty((0,T)\times Y);L^p(D))} \leqslant C. \tag{2.20}$$

However such *Lipschitz regularity* does not necessarily hold for solutions of the Burgers' equation. As is well-known, solutions to the Burgers' equation develop shocks and the derivative u_x blows up. Similar considerations also apply for higher derivatives of u in y. Hence, a bound like (2.20) does not hold in general for our underlying parameters to observable map . In other words, singularities in physical space propagate as singularities in parametric space.

On the other hand, if $p = 1$ in the above calculation, we see that the right hand side of (2.19) is zero and we integrate over space to obtain a bound on the total variation norm of u. This is not unexpected as solutions to Burgers' equation are total variation diminishing in space and this regularity is retained in the parametric space. In fact, one can make the above argument rigorous (by standard vanishing viscosity approximations) and extend to general scalar convection diffusion equations. We can prove the following theorem,

Theorem 2.3. *Consider the scalar conservation law, i.e parameterized convection-diffusion PDE* (2.1) *with $m = 1$ and $\nu = 0$, in domain $D = \mathbb{R}^{d_s}$ and assume that the initial data $u_0 \in L^\infty(D \times Y) \cap BV(D \times Y)$ and support of u_0 is compact. Assume, furthermore that $\psi \in L^\infty(D \times (0,T))$ and $g \in C^1(\mathbb{R})$. Then, the parameters to observable map $\mathcal{L}$, defined in* (2.3) *satisfies,*

$$\|\mathcal{L}\|_{BV(Y)} \leqslant C, \tag{2.21}$$

for some constant C, depending on the initial data u_0, ψ and g. Moreover, if the numerical solution u^Δ is generated by a monotone or TVD numerical method, then the approximate parameters to observable map $\mathcal{L}^\Delta$ (2.4) *satisfies*

$$\|\mathcal{L}^\Delta\|_{BV(Y)} \leqslant C, \tag{2.22}$$

It is unclear if bounds such as (2.21), (2.22) hold for systems of conservation laws or more general convection-diffusion equations such as the Navier-Stokes equations. Such bounds might be a reasonable assumption in many cases. However, the presence of instabilities/turbulence might lead to a blow up of the Total variation bound [15, 16]. In these cases, one can only prove a bound on the numerical approximation $\mathcal{L}^\Delta$ of the form,

$$\|\mathcal{L}^\Delta\|_{BV(Y)} \leqslant C\Delta^{-r}, \quad 0 \leqslant r < 1. \tag{2.23}$$

It is unclear if theorem 2.2 holds for BV functions with $s = 1$. Although it has not been rigorously shown, perusing the steps of the proofs of [36] strongly indicate that a theorem such as theorem 2.2 might indeed hold for BV functions.

It could be argued that estimates such as (2.21) are rather pessimistic as they seek to bound the observable $\mathcal{L}$ in terms of the underlying solution field $\mathbf{U}$, see (2.16), and might overestimate possible cancellations. This is indeed the case, at least for scalar conservation laws with random initial data i.e, (2.1) with $m = 1, \nu = 0$ and $\mathbf{F}(y, \mathbf{U}) = \mathbf{F}(\mathbf{U})$. In this case, we have the following theorem,

Theorem 2.4. *Consider the scalar conservation law, i.e parameterized convection-diffusion PDE* (2.1) *with* $m = 1$ *and* $\nu = 0$*, in domain* $D = \mathbb{R}^{d_s}$ *and time period* $[0,T]$ *and assume that the initial data* $u_0 \in W^{1,\infty}\left(Y; L^1(D)\right)$ *and support of* u_0 *is compact. Assume, furthermore that* $\psi \in L^\infty(D \times (0,T))$ *and* $g \in W^{1,\infty}(\mathbb{R})$*. Then, the parameters to observable map* $\mathcal{L}$*, defined in* (2.3) *satisfies,*

$$\|\mathcal{L}\|_{W^{1,\infty}(Y)} \leqslant C, \tag{2.24}$$

for some constant C*, depending on the initial data* u_0*,* ψ *and* g*. Moreover, if the numerical solution* u^Δ *is generated by a monotone numerical method, then the approximate parameters to observable map* $\mathcal{L}^\Delta$ (2.4) *satisfies*

$$\|\mathcal{L}^\Delta\|_{W^{1,\infty}(Y)} \leqslant C, \tag{2.25}$$

The proof of this theorem is a consequence of the L^1 stability (contractivity) of the solution operator for scalar conservation laws [10]. In fact, for any $y, y^* \in Y$, a straightforward calculation using the definition (2.3) and the Lipschitz regularity of g yields,

$$\begin{aligned}
|\mathcal{L}(y) - \mathcal{L}(y^*)| &\leqslant \|\psi\|_\infty \|g\|_{\mathrm{Lip}} \int_0^T \|u(t,.,y) - u(t,.,y^*)\|_1 dt, \\
&\leqslant \|\psi\|_\infty \|g\|_{\mathrm{Lip}} \int_0^T \|u_0(.,y) - u_0(.,y^*)\|_1 dt, \quad \text{by } L^1 - \text{contractivity} \\
&\leqslant \|\psi\|_\infty \|g\|_{\mathrm{Lip}} T \|u_0\|_{W^{1,\infty}(Y, L^1(D))} \leqslant C
\end{aligned}$$

The above proof also makes it clear that bounds such as (2.24) and (2.25) will hold for systems of conservation laws and the incompressible Navier-Stokes equations as long as there is some stability of the field with respect to the input parameter vector. On the other hand, we cannot expect any such bounds on the higher parametric derivatives of $\mathcal{L}$, due to the lack of differentiability of the underlying solution field. The above theorem also demonstrates that there is a possible gain in regularity (in terms of integrability) by considering an observable rather than the whole field.

Given a bound such as (2.24) or (2.25), we can illustrate the difficulty of approximating the map $\mathcal{L}$ by considering a prototypical problem, namely that of learning the lift and the drag of the RAE2822 airfoil (see section 5.2). For this problem, the underlying parameter space is six-dimensional. Assuming that $s = 1$ in theorem 2.2 and requiring that the approximation error is at most one percent relative error i.e $\varepsilon = 10^{-2}$ in (2.15), yields a neural network of size $O(10^{12})$ tunable parameters and at least 6 layers. Such a large network is clearly unreasonable as it will be very difficult to train and expensive to evaluate.

The above argument illustrates the difficulties of learning PDEs that arise in fluid dynamics. This is in contrast to elliptic and parabolic PDEs as solutions of these PDEs are much more regular and one can harness approximation theoretic arguments like theorem 2.2 to provide reasonable estimates on network sizes, see [43] for examples. On the other hand, approximation theory only serves to highlight that approximating our parameters to observable map $\mathcal{L}$ with neural networks is a priori, very difficult.

2.3.3 Trained networks and generalization error

We can argue that the bounds implied by approximation theory are not completely relevant in our context. After all, our aim is not to estimate the optimal approximation error of our map $\mathcal{L}$ with respect to neural networks, but rather to estimate the error that one makes while approximating $\mathcal{L}^\Delta$ with a *trained network* i.e, a network trained to minimize loss functions such as (2.12) with a stochastic gradient descent method. Thus, the relevant measure of appropriateness of a neural network in our context, is provided by the so-called *generalization error* i.e, once a (local) minimum θ^* of (2.12) has been computed, we are interested in the following error,

$$\mathcal{E}_G(\theta^*) := \left(\int_Y |\mathcal{L}^\Delta(y) - \mathcal{L}_{\theta^*}(y)|^p d\mu(y) \right)^{\frac{1}{p}}, \quad 1 \leqslant p \leqslant \infty. \tag{2.26}$$

In practice, one estimates the generalization error on a so-called *test set* i.e, $\mathcal{T} \subset Y$, with $\#(\mathcal{T}) >> N = \#(\mathcal{S})$. For instance, $\mathcal{T}$ could consist of i.i.d random points in Y, drawn from the underlying distribution μ. In this case, the generalization error (2.26) can be estimated by the *prediction error*,

$$\mathcal{E}_P(\theta^*) := \left(\frac{1}{\#(\mathcal{T})} \sum_{y \in \mathcal{T}} |\mathcal{L}^\Delta(y) - \mathcal{L}_{\theta^*}(y)|^p \right)^{\frac{1}{p}} \tag{2.27}$$

It turns out that one can estimate the generalization error (2.26) by using tools from machine learning theory [44] as,

$$\mathcal{E}_G(\theta^*) \leqslant \sqrt{\frac{U}{N}}, \tag{2.28}$$

with $N = \#(\mathcal{S})$ being the number of training samples. The numerator U is typically estimated in terms of the Rademacher complexity or the Vapnik-Chervonenkis (VC) dimension of the network, see [44] and references therein for definitions and estimates.

Unfortunately, such bounds on U are very pessimistic in practice and over estimate the generalization error by several (tens of) orders of magnitude [2]. Consequently, obtaining sharp bounds on U in (2.28) is widely considered to be a major challenge in theoretical machine learning, [52, 2, 35] and references therein. A heuristic rule of thumb for the upper bound of U is simply the size (number of tunable parameters) in the network [2]. Much sharper bounds have been obtained recently, in the context of classification, by [2]. Their bound is based on subtle but computable noise stability properties of ReLU networks, that lead to *compression* i.e a reduction in the effective number of tunable parameters in the network. Consequently, this compression needs to be checked a posteriori on each network and is difficult to estimate a priori. Sharp bounds are available for networks with a single hidden layer, for instance in [35] and more recently in [13].

For our purposes, we will assume that $U \sim c_1 + c_2 M$, with M being the number of tunable parameters in our network (network size). Here, c_1 is a threshold parameter that prevents us from considering networks of very small size, and c_2 is a compression parameter that measures the effective number of network parameters, as in [2].

Even if we make a rather stringent assumption of $U \sim \mathcal{O}(1)$, we see that achieving a relative generalization error of one percent requires us to set $N = 10^4$. This is an unrealistically high number of samples in our context. We recall that each training sample is evaluated by (possibly expensive) CFD forward solve of (2.1). Hence, we can generate around $10^2 - 10^3$ samples in reasonable computational time. However, these choices may lead to unacceptably large generalization (or prediction) errors of 30 to 100%. This illustrates the fundamental practical challenge that we face i.e, we are in a *data poor* regime and a priori, it is completely unclear as to whether approximating the map $\mathcal{L}$ by deep neural networks will be effective or not. This should be contrasted with the prototypical situation in computer science where the triumphs of deep learning have been in the context of problems with big data.

Summarizing, theoretical considerations outlined above indicate the challenges of *finding deep neural networks to accurately learn maps of low regularity in a data poor regime.* We illustrate these difficulties with a simple numerical example.

2.3.4 An illustrative numerical experiment.

In this experiment we consider the parameter space $Y = [0,1]^6$ and train neural networks to approximate the following two maps,

$$\mathcal{L}^1(y) = \sum_{i=1}^{6} \sin(4\pi y_i), \quad \mathcal{L}^2(y) = \sum_{i=1}^{6} \frac{1}{i^3} \sin(4\pi y_i). \tag{2.29}$$

Clearly both maps are linear combinations of sine functions and are infinitely differentiable. However, the amplitude of derivatives of each map can be rather high. On the other hand, the derivatives of the scaled sine $\mathcal{L}^2$ are smaller in amplitude for most of the dimensions. Hence, the scaled sine has some sparsity (with respect to dimensions).

We define a neural network of the form (2.6) and with the architecture specified in table 3. In order to generate the training set, we select the first 128 Sobol points in $[0,1]^6$ and sample the functions (2.29)

Map	Error (Mean) in %	Error (Std.) in %
$\sum_{i=1}^{6} \sin(4\pi x_i)$	133.95	417.83
$\sum_{i=1}^{6} \frac{\sin(4\pi x_i)}{i^3}$	43.66	41.26

Table 1: Relative percentage Prediction errors (2.27) with $p = 2$, for the trained neural networks in the sum of sines experiment (2.29).

at these points. The training is performed with ADAM algorithm and hyperparameters defined in table 5.

The result of the training is shown in figure 2 (Top row) and indicates that the loss function is reduced during training by two-three orders of magnitude for both functions. The performance of the neural network is ascertained by choosing the first 8192 Sobol points in $[0,1]^6$ as the test set $\mathcal{T}$. The results of evaluating the trained networks on the test set is shown are figure 2, in terms of a scatter plot with the x-axis representing the ground truth (function values) and the y-axis representing the values predicted by the trained neural networks. As seen in the figure, results with the trained networks are very bad, with a large scatter, indicating that the networks generalize poorly. This is further quantified in table 1 where we present the relative prediction error (2.27) as a percentage and also present the standard deviation (on the test set) of the prediction error. Clearly, the errors are unacceptably high, particularly for the unscaled sum of sines $\mathcal{L}^1$ in (2.29). The errors reduce by an order of magnitude for the scaled sum of sines $\mathcal{L}^2$, but are still very high. This example illustrates the difficulty to approximating (even very regular) functions, in moderate to high dimensions, by neural networks when the training set is relatively small.

2.3.5 Pragmatic choice of network size

Given the above discussion, the size of the network (and its architecture) is crucial. We cannot use approximation theory estimates to choose the network size as it leads to unreasonably large networks. Instead, we estimate network size based on the following very deep but easy to prove fact about training neural networks,

Lemma 2.5. *For the map $\mathcal{L}^\Delta$ and for the training set $\mathcal{S}$, with $\#(\mathcal{S}) = N$, there exists a weight vector $\hat{\theta} \in \Theta$, and a resulting neural network of the form $\mathcal{L}_\theta$, with $\Theta \subset \mathbb{R}^M$ and $M = O(d + 1 + N)$, such that the following holds,*

$$\mathcal{L}^\Delta(z) = \mathcal{L}_{\hat{\theta}}(z), \quad \forall\, z \in \mathcal{S}. \tag{2.30}$$

The proof is presented in [52]. The above lemma implies that there exists a neural network with weight vector $\hat{\theta}$ of size $O(d + 1 + N)$ such that the training error defined by,

$$\mathcal{E}_T(\hat{\theta}) := \frac{1}{\#(\mathcal{S})} \sum_{y \in \mathcal{S}} |\mathcal{L}^\Delta(y) - \mathcal{L}_{\hat{\theta}}(y)|^p = 0. \tag{2.31}$$

Thus, it is reasonable to expect that a network of size $O(d + N)$ can be trained by a gradient descent method to achieve a very low training error. This allows us to estimate the generalization error (2.26) by a crude upper bound on $U \sim O(d + N)$ in (2.26). Thus, we are clearly in the overparametrized regime with more parameters than samples and overfitting can be a serious issue [44].

It is customary to monitor the generalization capacity of a trained neural network by computing a so-called *validation set* $\mathcal{V} \subset Y$ with $\mathcal{V} \cap \mathcal{S} = \Phi$, and evaluating the so-called validation loss,

$$J_{\mathcal{V}}(\theta) := \frac{1}{\#(\mathcal{V})} \sum_{y \in \mathcal{V}} \|\mathcal{L}^\Delta(y) - \mathcal{L}_\theta(y)\|_p^p. \tag{2.32}$$

A low validation loss is observed to correlate with low generalization errors. In order to generate the validation set, one can set aside a small proportion, say 10 − 20% of the training set as the validation set.

1. Number of hidden layers $(K-1)$
2. Number of units in k-th layer (d_k)
3. Exponent p in the loss function (2.11).
4. Exponent q in the regularization term in (2.12)
5. Value of regularization parameter λ in (2.12)
6. Choice of optimization algorithm (optimizer) – either standard SGD or ADAM.
7. Initial guess $\bar{\theta}$ in the SGD method (2.13).

Table 2: Hyperparameters in the training algorithm

2.4 Hyperparameters and Ensemble training.

The theoretical considerations in the last subsection clearly bring out the challenges in approximating the parameters to observable map $\mathcal{L}$ i.e low regularity and availability of relatively few training samples. However, these considerations might well be pessimistic due to the following factors,

- It must be remembered that bounds such as (2.21), (2.22) are upper bounds. The parameters to observable map $\mathcal{L}$ or $\mathcal{L}^\Delta$ may be far better behaved than a generic BV or Lipschitz function, on account of the averaging in defining $\mathcal{L}$ in (2.2). Moreover, the map might possess some hitherto unknown compression properties that will certainly reduce the upper bound U in (2.26). Hence, the generalization error might be significantly lower than what we have estimated.
- There are several hyperparameters involved in the training process. These include network size, specifics of the loss function and of the optimization algorithm. Although determining the hyperparamaters could be delicate as the performance of the network might rely on them sensitively, it also provides an impetus for finding the best hyperparameter configurations for our problem.

We address the second issue raised above, by devising a simple yet effective *ensemble training algorithm*. To this end, we consider the set of hyperparameters, listed in table 2. A few comments regarding the listed hyperparameters are in order. We need to chose the number of layers and width of each layer such that the overall network size, given by (2.10) is $O(N+d)$. The exponents p, q in the loss function and regularization terms (2.12) usually take the values of 1 or 2 and the regularization parameter λ is required to be small. The starting value $\bar{\theta}$ is chosen randomly.

For each choice of the hyperparameters, we realize a single sample in the hyperparameter space. Then, the machine learning algorithm 2.1 is run with this sample and the resulting loss function minimized.

Each sample in the hyperparameter space can be trained in parallel. Once the training for all the samples in this hyperparameter ensemble is complete, we can evaluate the prediction error (2.27) and calculate the best performing configuration, namely the one that minimizes the prediction error, in the hyperparameter space as well as the distribution of the prediction error with respect to the choice of hyperparameters. Further details of this procedure are provided in section 4.2. We remark that this procedure has many similarities to the *active learning* procedure proposed recently in [53].

3 Uncertainty quantification in CFD with deep learning.

We will apply the trained deep neural networks that approximate the parameters to observable map $\mathcal{L}^\Delta$, to efficiently quantify uncertainty in the underlying map. In particular, we are interested in estimating statistical moments of the parameters to observable map $\mathcal{L}$ of the form,

$$\overline{h} := \mathbb{E}_\mu(h(\mathcal{L}(y)) = \int_Y h(\mathcal{L}(y))d\mu(y), \tag{3.1}$$

for any $h \in \mathrm{Lip}(\mathbb{R}, \mathbb{R})$. Different functions h yield different statistical moments for $\mathcal{L}$.

As the map $\mathcal{L}$ is not directly accessible, we will approximate the statistical quantity of interest (3.1) with

$$\overline{h}^\Delta := \mathbb{E}_\mu(h(\mathcal{L}^\Delta(y)) = \int_Y h(\mathcal{L}^\Delta(y))d\mu(y). \tag{3.2}$$

Here, $\mathcal{L}^{\Delta}$ (2.4) is the numerical surrogate that approximates $\mathcal{L}$ to a given tolerance ε. Using (2.5), it is straightforward to verify that the *deterministic bias*,

$$|\overline{h} - \overline{h}^{\Delta}| \sim \varepsilon. \tag{3.3}$$

For notational simplicity, we assume that the underlying distribution on Y is uniform. Henceforth, $d\mu(y) = dy$.

The (stochastic) integral (3.2) needs to be approximated by a suitable (high-dimensional) quadrature rules such as,

3.1 Monte Carlo (MC) methods.

3.1.1 Baseline Monte Carlo (MC).

The Monte Carlo algorithm ([8] and references therein) consists of selecting J samples i.e points $\{y_j\}$ with each $y_j \in Y$ and $1 \leqslant j \leqslant J$, that are independent and identically distributed (*iid*). Then, the Monte Carlo approximation of the integral (3.2) is

$$\overline{h}_{mc} = \frac{1}{J} \sum_{j=1}^{J} h(\mathcal{L}^{\Delta}(y_j)). \tag{3.4}$$

Note that we suppress the dependence of the right hand side of (3.4) on the probability space for notational simplicity. It is well known that the root mean square error in approximating the integral (3.2), defined as

$$E_{rms}\left(|\overline{h}^{\Delta} - \overline{h}_{mc}|\right) := \mathbb{E}\left(|\overline{h}^{\Delta} - \overline{h}_{mc}|^2\right)^{\frac{1}{2}}, \tag{3.5}$$

is estimated by [8],

$$E_{rms}\left(|\overline{h}^{\Delta} - \overline{h}_{mc}|\right) \leqslant \sqrt{\frac{\mathbb{V}\left(h^{\Delta}\right)}{J}}. \tag{3.6}$$

Here, we denote the *variance* as,

$$\mathbb{V}\left(h^{\Delta}\right) = \mathbb{E}\left(\left(h^{\Delta} - \overline{h}^{\Delta}\right)^2\right), \quad h^{\Delta}(y) = h(\mathcal{L}^{\Delta}(y)), \quad \forall y \in Y, \tag{3.7}$$

and the mean $\overline{h}^{\Delta}$ is defined in (3.2).

To obtain a root mean square (rms) error of size ε, we observe from (3.6) that one has to set,

$$J \sim \frac{\mathbb{V}(h^{\Delta})}{\varepsilon^2},$$

We denote the cost of computing a single realization of $h(\mathcal{L}^{\Delta}(y_j))$, for any $1 \leqslant j \leqslant J$, by $\mathcal{C}$. This cost comprises of the cost of performing a high-resolution (at mesh size Δ) CFD simulation and the cost of computing quadratures to evaluate (2.2) and is typically of size $O\left(\Delta^{-\frac{1}{d_s+1}}\right)$.

Consequently, the total cost of the standard Monte Carlo algorithm to obtain an rms error of size ε is

$$\mathcal{C}_{mc} = J\mathcal{C} \sim \frac{\mathcal{C}\mathbb{V}(h^{\Delta})}{\varepsilon^2}. \tag{3.8}$$

We observe from (3.8) that if the variance is $O(1)$, then the cost of computing Monte Carlo simulations for UQ is prohibitively expensive as a large number of very expensive high-resolution CFD simulations need to be performed in order to achieve a tolerable error. Hence, it is imperative to seek more efficient alternatives to the baseline MC algorithm.

3.1.2 Deep learning Monte Carlo (DLMC)

Deep neural networks (2.6), even of large size, are very cheap to compute. This cost is expected to be significantly smaller than the underlying CFD solver and we wish to exploit this cost differential in order to accelerate Monte Carlo. We propose the following algorithm,

Algorithm 3.1. Deep learning Monte Carlo (DLMC).

Inputs: *Parameterized PDE* (2.1)*, Observables* (2.2)*, high-resolution numerical method for solving* (2.1) *and calculating* (2.2)*, function h in* (3.1)*.*

Goal: *Compute statistical quantity of interest* (3.2)*.*

Step 1: *Choose N iid random samples $\mathcal{S} = \{y_i\}$ with $y_i \in Y$ and $1 \leqslant i \leqslant N$. With $\mathcal{S}$ as training set, compute an optimal neural network $\mathcal{L}^*$ for the parameters to observable map $\mathcal{L}^\Delta$, by using algorithm 2.1.*

Step 2: *Choose J_L iid random samples $\{y_j\}$ with $y_j \in Y \setminus \mathcal{S}$ and $1 \leqslant j \leqslant J_L$.*

Step 3: *For each y_j, evaluate $\mathcal{L}^*(y_j)$ by computing the neural network $\mathcal{L}^*$.*

Step 4: *Approximate the statistical quantity of interest* (3.2) *by*

$$\overline{h}^*_{mc} = \frac{1}{J_L} \sum_{j=1}^{J_L} h(\mathcal{L}^*(y_j)). \tag{3.9}$$

Thus, we replace high resolution CFD simulations with (much cheaper) neural networks in evaluating the Monte Carlo samples, while still using the (much lower number of) high resolution simulations to train the neural network.

In order to perform the error and complexity analysis of the DLMC algorithm 3.1, we need to define,

$$\overline{h}^* := \mathbb{E}(h(\mathcal{L}^*(y)) = \int\limits_Y h(\mathcal{L}^*(y))dy. \tag{3.10}$$

It is straightforward to calculate that the *training bias*, $|\overline{h}^\Delta - \overline{h}^*|$ can be estimated by

$$|\overline{h}^\Delta - \overline{h}^*| \sim \mathcal{E}_G, \tag{3.11}$$

with $\mathcal{E}_G$ denoting the generalization error (2.26). Note the dependence of the generalization error on the local optimum θ^*, that we suppress for notational simplicity.

We have the following theorem for complexity of the DLMC method,

Theorem 3.2. *Let $\overline{h}^\Delta$ be as defined in* (3.2)*. For any given tolerance $\varepsilon > 0$, assume that a neural network $\mathcal{L}^*$ can be found with algorithm 2.1 such that the generalization error $\mathcal{E}_G$* (2.26)*. satisfies*

$$\mathcal{E}_G \sim \varepsilon. \tag{3.12}$$

Then, the speed up Σ_{dlmc}, defined as the ratio of the cost of baseline MC algorithm of the previous subsection and the cost of the DLMC algorithm 3.1 for computing the statistical quantity of interest $\overline{h}^\Delta$ to an error of size $O(\varepsilon)$ is given by,

$$\frac{1}{\Sigma_{dlmc}} \sim \frac{\varepsilon^2 N}{\mathbb{V}(h^\Delta)} + \frac{\mathcal{C}_*}{\mathcal{C}}, \tag{3.13}$$

with $\mathcal{C}, \mathcal{C}_*$ being the computational cost of computing $\mathcal{L}^\Delta(y)$ (high resolution CFD simulation) and $\mathcal{L}^*(y)$ (deep neural network) for any $y \in Y$, N being the number of training samples i.e $\#(\mathcal{S})$ in algorithm 3.1 and $\mathbb{V}$ denoting the variance defined in (3.7)

Proof. Let $h_*(y) = h(\mathcal{L}_*(y))$ for all $y \in Y$. We observe from (3.11) and assumption (3.12) that the *training bias* satisfies $|\overline{h}^{\Delta} - \overline{h}^{*}| \sim \varepsilon$. Hence, it is enough to consider the following root mean square (stochastic) error,

$$E^*_{rms}\left(|\overline{h}^* - \overline{h}^*_{mc}|\right) := \mathbb{E}\left(|\overline{h}^* - \overline{h}^*_{mc}|^2\right)^{\frac{1}{2}}, \tag{3.14}$$

Following [8], this error can be estimated as

$$E^*_{rms}\left(|\overline{h}^* - \overline{h}^*_{mc}|\right) \leqslant \sqrt{\frac{\mathbb{V}(h^*)}{J_L}}. \tag{3.15}$$

Here, we denote the *variance* with respect to the neural network as,

$$\mathbb{V}(h^*) = \mathbb{E}\left(\left(h^* - \overline{h}^*\right)^2\right), \tag{3.16}$$

and the mean $\overline{h}^*$ is defined in (3.10).

In order to obtain a root mean square (rms) error of size ε, we observe from (3.15) that one has to set,

$$J_L \sim \frac{\mathbb{V}(h^*)}{\varepsilon^2},$$

Under the assumptions that the actual cost of training by (stochastic) gradient descent in algorithm 2.1 is significantly smaller than the cost of generating the training data by high-resolution CFD simulations, the total cost of the DLMC algorithm for computing (3.2) to an error of size $O(\varepsilon)$ is given by

$$\begin{aligned} \mathcal{C}_{dlmc} &= \mathcal{C}N + \mathcal{C}_* J_L, \\ &= \mathcal{C}N + \mathcal{C}_* \frac{\mathbb{V}(h^*)}{\varepsilon^2}. \end{aligned} \tag{3.17}$$

By dividing the above expression with the cost of the baseline Monte Carlo algorithm (3.8) yields,

$$\begin{aligned} \frac{1}{\Sigma_{dlmc}} &= \frac{\mathcal{C}_{dlmc}}{\mathcal{C}_{mc}} \\ &\sim \frac{\varepsilon^2 N}{\mathbb{V}(h^\Delta)} + \frac{\mathcal{C}_* \mathbb{V}(h^*)}{\mathcal{C}\mathbb{V}(h^\Delta)} \\ &\sim \frac{\varepsilon^2 N}{\mathbb{V}(h^\Delta)} + \frac{\mathcal{C}^*}{\mathcal{C}}, \end{aligned}$$

thus, yielding the claimed speedup (3.13). Note that in the last line of the above calculation, we have used a straightforward estimate of the form $|\mathbb{V}(h^\Delta) - \mathbb{V}(h^*)| \sim \varepsilon$, derived by applying the assumption on generalization error (3.12). □

By using the fact that the number of samples in the baseline MC algorithm is $J \sim \frac{\mathbb{V}(h^\Delta)}{\varepsilon^2}$, we can write the speed up estimate (3.13) as,

$$\frac{1}{\Sigma_{dlmc}} \sim \frac{N}{J} + \frac{\mathcal{C}^*}{\mathcal{C}}, \tag{3.18}$$

Remark 3.3. In particular, we know that the cost $\mathcal{C} = O\left(\Delta^{-\frac{1}{d+1}}\right)$ is very high for a small mesh size Δ. On the other hand the cost $\mathcal{C}_* = O(M)$ with M being the number of neurons (2.10) in the network. This cost is expected to be much lower than $\mathcal{C}$. As long as we can guarantee that $N << J$ in (3.18), it follows that the speedup Σ_{dlmc}, which measures the gain of using the deep learning based DLMC algorithm 3.1 over the baseline Monte Carlo method, can be quite significant. ■

Remark 3.4. Using the estimate (2.28) on the generalization error and the assumption (3.12), we can rewrite the speedup with respect ot the DLMC algorithm as,

$$\frac{1}{\Sigma_{dlmc}} \sim \frac{U}{\mathbb{V}(h^{\Delta})} + \frac{\mathcal{C}^*}{\mathcal{C}}, \tag{3.19}$$

Thus, a speed up is guaranteed if $U < \mathbb{V}(h^{\Delta})$. Hence, the compression properties of the underlying maps and the resulting trained networks are crucial in ensuring a speedup with the DLMC method over the baseline MC algorithm. ■

3.2 Quasi-Monte Carlo (QMC) methods

3.2.1 Baseline QMC algorithm.

Quasi-Monte Carlo (QMC) methods are deterministic quadrature rules for computing integrals, [8] and references therein. The key idea underlying QMC is to choose a (deterministic) set of points on the domain of integration, designed to achieve some measure of *equidistribution*. Thus, the QMC algorithm is expected to approximate integrals with higher accuracy than MC methods, whose nodes are randomly chosen [8].

For any statistical quantity of interest $\overline{h}$ (3.1) and any tolerance $\varepsilon > 0$, we can choose a mesh size Δ small enough in order to ensure that the deterministic bias (3.3) is of size $O(\varepsilon)$. Thus, we will approximate the integral (3.2) on the integration domain Y. To this end, we choose a set of points $\mathcal{J}_q = \{y_j\} \subset Y$ with $1 \leqslant j \leqslant J_q = \#(\mathcal{J}_q)$. Then, the integral (3.2) is approximated by the sum,

$$\overline{h}_{qmc} := \frac{1}{J_q} \sum_{j=1}^{J_q} h(\mathcal{L}^{\Delta}(y_j)), \quad \forall\ y_j \in \mathcal{J}_q \tag{3.20}$$

Following [8], one can estimate the error in integrating (3.2) by the sum (3.20) in terms of the so-called Koksma-Hlawka inequality,

$$|\overline{h}^{\Delta} - \overline{h}_{qmc}| \sim \mathcal{V}_r(h^{\Delta})\mathcal{D}(\mathcal{J}_q). \tag{3.21}$$

Here, $\mathcal{V}_r(h^{\Delta})$ is some measure of *variation* of the integrand $h(\mathcal{L}^{\Delta})$ in (3.2). It can be bounded above as

$$\mathcal{V}_r(h^{\Delta}) \leqslant C \int_Y \left| \frac{\partial^d}{\partial y_1 y_2 \cdots y_d} h(\mathcal{L}^{\Delta}(y)) dy \right|, \tag{3.22}$$

by the Hardy-Krause variation of the function $\mathcal{L}^{\Delta}$. This upper bound requires some regularity for the underlying map in terms of mixed partial derivatives. However, it is well-known that the bound (3.22) is a gross overestimate of the integration error [8]. Hence, it serves only as a sufficient condition in the Koksma-Hlawka inequality (3.21).

The $\mathcal{D}(\mathcal{J}_q)$ in (3.21) is the so-called *discrepancy* of the point sequence $\mathcal{J}_q$, defined formally in [8] and references therein. It measures how equally is the point sequence $\mathcal{J}_q$ spread out in Y. The whole objective in the development of quasi-Monte Carlo (QMC) methods is to design *low discrepancy sequences*. In particular, there exists many popular QMC sequences such as Sobol, Halton or Niederreiter [8] that satisfy the following bound on discrepancy,

$$\mathcal{D}(\mathcal{J}_q) \sim \frac{(\log(J_q))^d}{J_q}. \tag{3.23}$$

Thus, the integration error with QMC quadrature based on these rules behaves log-linearly with respect to the number of quadrature points. Ignoring the logarithmic terms by assuming that the dimension d is not too high, we observe that the number of quadrature points needed within a QMC method to achieve an error of size size $O(\varepsilon)$ for integrating (3.2), is

$$J_q \sim \frac{\mathcal{V}_r(h^{\Delta})}{\varepsilon}. \tag{3.24}$$

Consequently, the total cost of a QMC approximation of (3.2), to achieve integration error of size $O(\varepsilon)$ is given by,

$$\mathcal{C}_{qmc} \sim \frac{\mathcal{V}_r(h^\Delta)}{\varepsilon}\mathcal{C}, \tag{3.25}$$

with $\mathcal{C}$ representing the computational cost of evaluating (3.2) for any $y \in Y$ by a high-resolution CFD simulation of (2.1).

Comparing this cost with the cost of the corresponding MC method (3.8), we notice that there can considerable speed up with the QMC method as long as $\mathcal{V}_r(h^\Delta) \sim \mathbb{V}(h^\Delta)$. This assumption implies a certain amount of regularity in the underlying parameters to observable map $\mathcal{L}$. Moreover, the deterministic pointwise integration error (3.21) can be very different from the probabilistic root mean square (rms) error (3.6). However, as a rule of thumb, it is reasonable to expect that the QMC method significantly outperforms the MC method as long as there is some regularity in the underlying integrand and if the dimension d of the domain of integration, is not very high. Hence, QMC is an widely used alternative to MC methods in the context of UQ, see [20] and references therein and more recently to a forthcoming paper [30] in the context of UQ for CFD.

3.2.2 Deep learning quasi-Monte Carlo (DLQMC)

Even if QMC is more efficient than the MC method, it is clear from (3.25) that a large number of expensive CFD forward solves have to be performed to achieve reasonably low error. Next, we describe an algorithm that combines the QMC method with a deep neural network approximation of the underlying parameters to observable map.

Algorithm 3.5. Deep learning Quasi-Monte Carlo (DLQMC).

Inputs: *Parameterized PDE* (2.1), *Observables* (2.2), *high-resolution numerical method for solving* (2.1) *and calculating* (2.2), *function h in* (3.1).

Goal: *Compute statistical quantity of interest* (3.2).

Step 1: *For $J_L \in \mathbb{N}$, select $\mathcal{J}_L = \{y_j\} \subset Y$, with $1 \leqslant j \leqslant J_L$, as the first J_L points from a Quasi-Monte Carlo low-discrepancy sequence such as Sobol, Halton etc.*

Step 2; *For some $N << J_L$, denote $\mathcal{S} = \{y_j\} \subset \mathcal{J}_L$, with $1 \leqslant j \leqslant N$ i.e, first N points in $\mathcal{J}_L$, and evaluate the map $h(\mathcal{L}^\Delta(y_j))$, for all $y_j \in \mathcal{J}_d$ with the high-resolution CFD simulation. As a matter of fact, any consecutive set of N QMC points can serve as the training set.*

Step 3: *With $\mathcal{S}$ as the training set, compute an optimal neural network $\mathcal{L}^*$ for the parameters to observable map $\mathcal{L}^\Delta$, by using algorithm 2.1.*

Step 4: *Approximate the statistical quantity of interest* (3.2) *by*

$$\overline{h}^*_{qmc} = \frac{1}{J_L}\left(\sum_{j=1}^{N} h(\mathcal{L}^\Delta(y_j)) + \sum_{j=N+1}^{J_L} h(\mathcal{L}^*(y_j))\right) \tag{3.26}$$

We remark that the main difference in the structure of the DLQMC algorithm and the DLMC algorithm 3.1 lies in the fact that the training samples can be reused in the QMC estimator, unlike in the Monte Carlo method. The complexity of this algorithm is described in the following theorem,

Theorem 3.6. *Let $\overline{h}^\Delta$ be as defined in* (3.2)*. For any given tolerance $\varepsilon > 0$ and under the assumption that the neural network $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ has a generalization error* (2.26) *of size $O(\varepsilon)$, the speed up Σ_{dlqmc}, defined as the ratio of the cost of baseline QMC algorithm and the cost of the DLQMC algorithm 3.5 for computing the statistical quantity of interest $\overline{h}^\Delta$ to an error size of $O(\varepsilon)$ is given by,*

$$\frac{1}{\Sigma_{dlqmc}} \sim \left(1 - \frac{\mathcal{C}_*}{\mathcal{C}}\right)\frac{\varepsilon N}{\mathcal{V}_r(h^\Delta)} + \frac{\mathcal{C}_*}{\mathcal{C}}\frac{\mathcal{V}_r(h^*)}{\mathcal{V}_r(h^\Delta)} \tag{3.27}$$

with $\mathcal{C}, \mathcal{C}_*$ *being the computational cost of computing* $\mathcal{L}^\Delta(y)$ *(high resolution CFD simulation) and* $\mathcal{L}^*(y)$ *(deep neural network) for any* $y \in Y$*,* h^* *defined in* (3.10) *and* $\mathcal{V}_r(f)$ *is the variation of any integrand* f*, that appears in the Koksma-Hlawka inequality* (3.21).

Proof. The QMC estimator (3.26) can be rewritten as

$$\overline{h}^*_{qmc} = \frac{1}{J_L}\left(\sum_{j=1}^{N}\left(h(\mathcal{L}^\Delta(y_j)) - h(\mathcal{L}^*(y_j))\right) + \sum_{j=1}^{J_L} h(\mathcal{L}^*(y_j))\right).$$

With $\bar{h}^\Delta, \bar{h}^*$ defined in (3.2), (3.10), we have the following calculation,

$$\begin{aligned} |\bar{h}^\Delta - \bar{h}^*_{qmc}| &= \left|\bar{h}^\Delta - \bar{h}^* + \bar{h}^* - \bar{h}^*_{qmc}\right| \\ &\leqslant \underbrace{\left|\bar{h}^\Delta - \bar{h}^*\right|}_{T_1} + \underbrace{\frac{1}{J_L}\sum_{j=1}^{N}\left|\left(h(\mathcal{L}^\Delta(y_j)) - h(\mathcal{L}^*(y_j))\right)\right|}_{T_2} + \underbrace{\left|\bar{h}^* - \sum_{j=1}^{J_L} h(\mathcal{L}^*(y_j))\right|}_{T_3}, \end{aligned}$$

By the assumption that the generalization error (2.26) is $O(\varepsilon)$, we have

$$T_1 \leqslant \|h\|_{Lip}\mathcal{E}_G \sim \varepsilon.$$

Similarly,

$$T_2 \leqslant \frac{\mathcal{E}_T}{J_L} \sim \varepsilon.$$

The above estimate follows as the training error (2.31) is certainly of size $O(\varepsilon)$. Finally, by the Koksma-Hlawka inequality (and ignoring the logarithmic terms for simplicity of the exposition), we have

$$T_3 \sim \frac{\mathcal{V}_r(h^*)}{J_L}.$$

Thus, in order to obtain an overall error of $O(\varepsilon)$, we require $J_L \sim \frac{\mathcal{V}_r(h^*)}{\varepsilon}$. Hence, after assuming that the training costs are negligible compared to the cost of generating the training set, the total cost for the DLQMC algorithm 3.5 in achieving an error of size ε is given by,

$$\mathcal{C}_{dlqmc} \sim \mathcal{C}N + \mathcal{C}_*(J_L - N) \sim (\mathcal{C} - \mathcal{C}_*)N + \mathcal{C}_*\frac{\mathcal{V}_r(h^*)}{\varepsilon}. \tag{3.28}$$

Dividing the above with the cost of the baseline QMC algorithm (3.25) yields (3.27).

□

Remark 3.7. One can use (3.24) and rewrite the speed up as

$$\frac{1}{\Sigma_{dlqmc}} \sim \left(1 - \frac{\mathcal{C}_*}{\mathcal{C}}\right)\frac{N}{J_q} + \frac{\mathcal{C}_*}{\mathcal{C}}\frac{\mathcal{V}_r(h^*)}{\mathcal{V}_r(h^\Delta)} \tag{3.29}$$

Given that $\mathcal{C}_* << \mathcal{C}$ and $\mathcal{V}_r(h^*) \sim \mathcal{V}_r(h^\Delta)$, we expect a significant speed up with the DLQMC algorithm over the baseline QMC algorithm, as long as $N << J_q$. ■

Remark 3.8. Using (2.28), we can rewrite the speedup estimate as

$$\frac{1}{\Sigma_{dlqmc}} \sim \left(1 - \frac{\mathcal{C}_*}{\mathcal{C}}\right)\frac{U}{\varepsilon\mathcal{V}_r(h^\Delta)} + \frac{\mathcal{C}_*}{\mathcal{C}}\frac{\mathcal{V}_r(h^*)}{\mathcal{V}_r(h^\Delta)} \tag{3.30}$$

Hence, a speed up is guaranteed as long as $U \leqslant \varepsilon\mathcal{V}_r(h^\Delta)$. This puts a constraint on the compression properties of the trained networks. Comparing with the DLMC speedup estimate (3.19), we see that as long as $\mathcal{V}_r(h^\Delta) \sim \mathbb{V}(h^\Delta)$, it is harder to obtain a speedup with the DLQMC method over the baseline QMC method than it is with the DLMC method over the baseline MC method. This is not surprising as QMC converges faster than MC. On the other hand, it could happen for maps of low regularity (as in our case) that $\mathcal{V}_r(h^\Delta) >> \mathbb{V}(h^\Delta)$. This would result in a better speedup with the DLQMC algorithm than with the DLMC algorithm. ■

Remark 3.9. The estimator (3.26) can be modified to

$$\overline{h}^*_{qmc} = \frac{1}{J_L}\sum_{j=1}^{J_L} h(\mathcal{L}^*(y_j)) \tag{3.31}$$

In this case, we do not use the training samples in the estimator and the proof of theorem 3.6 is readily modified to obtain a speedup estimate,

$$\frac{1}{\Sigma_{dlqmc}} \sim \frac{N}{J_q} + \frac{\mathcal{C}_*}{\mathcal{C}}\frac{\mathcal{V}_r(h^*)}{\mathcal{V}_r(h^\Delta)} \tag{3.32}$$

Comparing with the speedup estimate (3.29), we observe a minor difference of the factor $1 - \frac{\mathcal{C}_*}{\mathcal{C}}$. As $\mathcal{C}_* << \mathcal{C}$, we see that estimator (3.31) can also be used with the same effect as the estimator (3.26). ■

3.3 Computation of probability distributions

In the context of UQ, we have to define appropriate statistical quantities of interest in (3.2) by specifying h. By letting $h(w) = w$, we compute the mean of the observable $\mathcal{L}$. Higher moments can be defined analogously. Moreover, we are also interested in the entire measure (probability distribution) of the observable. To this end, we assume that the underlying probability distribution on the input parameters to the problem (2.1) is given by $\mu \in \text{Prob}(Y)$. In the context of forward UQ, we are interested in how this initial measure is changed by the parameters to observable map $\mathcal{L}$ (or its high-resolution numerical surrogate $\mathcal{L}^\Delta$). Hence, we are interested in the *push forward* measure $\hat{\mu}^\Delta \in \text{Prob}(\mathbb{R})$ given by

$$\hat{\mu}^\Delta := \mathcal{L}^\Delta \# \mu, \quad \Rightarrow \quad \int_{\mathbb{R}} f(z)d\hat{\mu}^\Delta(z) = \int_Y f(\mathcal{L}^\Delta(y))d\mu(y), \tag{3.33}$$

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

Note that the measure $\hat{\mu}^\Delta$ contains all the statistical information on the observable $\mathcal{L}^\Delta$. Any moment can be computed by integrating an appropriate test function with respect to this measure. However, this measure is not available analytically and needs to approximated. *The task of efficiently computing this probability distribution is significantly harder than just estimating the mean and variance of the underlying map.*

We can adapt the baseline MC or QMC algorithm to approximate this measure. For definiteness, we consider the case of the baseline QMC algorithm here. In this context, the measure $\hat{\mu}^\Delta$ can be approximated by,

$$\hat{\mu}_{qmc} = \frac{1}{J_q}\sum_{j=1}^{J_d} \delta_{\mathcal{L}^\Delta(y_j)} \quad \Rightarrow \quad \int_{\mathbb{R}} f(z)d\hat{\mu}_{qmc}(z) = \frac{1}{J_q}\sum_{j=1}^{J_d} f\left(\mathcal{L}^\Delta(y_j)\right). \tag{3.34}$$

Here y_j are the first J_q QMC quadrature points described in the previous section. We want to estimate the difference between the measures $\hat{\mu}^\Delta$ and $\hat{\mu}_{qmc}$ and need some metric on the space of probability measures, to do so. One such metric is the so-called Wasserstein metric [50]. To define this metric, we need the following,

Definition 3.10. Given two measures, $\nu, \sigma \in \text{Prob}(\mathbb{R})$, a transport plan $\pi \in \text{Prob}(\mathbb{R}^2)$ is a probability measure on the product space such that the following holds for all measurable functions F, G,

$$\int_{\mathbb{R}\times\mathbb{R}} (F(u) + G(v))d\pi(u,v) = \int_{\mathbb{R}} F(u)d\nu(u) + \int_{\mathbb{R}} G(v)d\sigma(v). \tag{3.35}$$

The set of transport plans is denoted by $\Pi(\nu, \sigma)$ ■

Then for $1 \leqslant p < \infty$, the p-Wasserstein distance [50] between $\hat{\mu}^\Delta$ and $\hat{\mu}_{qmc}$ is defined as,

$$W_p\left(\hat{\mu}^\Delta, \hat{\mu}_{qmc}\right) := \left(\inf_{\pi \in \Pi\left(\hat{\mu}^\Delta, \hat{\mu}_{qmc}\right)} \int_{\mathbb{R}\times\mathbb{R}} |u - v|^p d\pi(u,v)\right)^{\frac{1}{p}} \tag{3.36}$$

Although it is very difficult to prove (due to technical reasons), it is fair to surmise on the basis of numerical evidence (see section 5) that the error in the Wasserstein metric behaves as,

$$W_p\left(\hat{\mu}^{\Delta}, \hat{\mu}_{qmc}\right) \sim \left(\frac{V^{\Delta}}{J_d}\right)^{\alpha}, \tag{3.37}$$

for some $0 < \alpha \leqslant 1$ and $V^{\Delta} = V(\mathcal{L}^{\Delta})$ is some measure of the variation of the underlying integrand.

Requiring that the Wasserstein distance is of size $O(\varepsilon)$ for some tolerance $\varepsilon > 0$, entails choosing $J_q \sim \frac{V^{\Delta}}{\varepsilon^{\frac{1}{\alpha}}}$ and leads to a cost of

$$\hat{C}_{qmc} \sim \mathcal{C}\frac{V^{\Delta}}{\varepsilon^{\frac{1}{\alpha}}}, \tag{3.38}$$

with $\mathcal{C}$ being the computational cost of a single CFD forward solve.

We can readily adapt the DLQMC algorithm 3.5 to approximate the measure $\hat{\mu}^{\Delta}$ by changing the estimator in Step 4 of this algorithm to,

$$\hat{\mu}^{*}_{qmc} = \frac{1}{J_L}\sum_{j=1}^{J_L} \delta_{\mathcal{L}^{*}(y_j)}, \tag{3.39}$$

The consequent gain in efficiency is quantified in the following theorem,

Theorem 3.11. *Under the assumption that the generalization error* (2.26) *is of size $O(\varepsilon)$ and the baseline QMC estimator* (3.34) *follows the error estimate* (3.37) *and with the notation of Theorem 3.6, the speedup with the DLQMC algorithm 3.5 in approximating the measure $\hat{\mu}^{\Delta}$ to $O(\varepsilon)$ satisfies,*

$$\frac{1}{\Sigma_{dlqmc}} \sim \frac{N}{J_q} + \frac{\mathcal{C}_*}{\mathcal{C}}\frac{V^*}{V^{\Delta}}. \tag{3.40}$$

Here, $V^* = V(\mathcal{L}^*)$ is an estimate on the variation that arises in a QMC estimate such as (3.37).

Proof. We define the measure $\hat{\mu}^* = \mathcal{L}^*\#\mu \in \text{Prob}(\mathbb{R})$ and estimate,

$$W_p\left(\hat{\mu}^{\Delta}, \hat{\mu}^{*}_{qmc}\right) \leqslant W_p\left(\hat{\mu}^{\Delta}, \hat{\mu}^{*}\right) + W_p\left(\hat{\mu}^{*}, \hat{\mu}^{*}_{qmc}\right).$$

We claim that assuming that the generalization error $\mathcal{E}_G \sim \varepsilon$ implies that $W_p\left(\hat{\mu}^{\Delta}, \hat{\mu}^{*}\right) \sim \varepsilon$. To see this, we define a transport plan $\pi^* \in \text{Prob}(\mathbb{R}^2)$ by $\pi^* = \hat{\mu}^{\Delta} \otimes \hat{\mu}^*$. Then,

$$\begin{aligned}
\int_{\mathbb{R}^2} |u-v|^p \, d\pi^*(u,v) &= \int_{\mathbb{R}^2} |u-v|^p \, d(\mathcal{L}^{\Delta}\#\mu \times \mathcal{L}^*\#\mu)(u,v), \\
&= \int_Y |\mathcal{L}^{\Delta}(y) - \mathcal{L}^*(y)|^p \, d\mu(y) := \mathcal{E}_G^p \\
&\sim \varepsilon^p.
\end{aligned}$$

This provides an upper bound on the Wasserstein distance (3.36) and proves the claim. Similarly, we use an estimate of the type (3.37) to obtain $W_p\left(\hat{\mu}^{*}, \hat{\mu}^{*}_{qmc}\right) \sim \left(\frac{V^*}{J_L}\right)^{\alpha}$. Requiring this error to be $O(\varepsilon)$ yields $J_L \sim \frac{V^*}{\varepsilon^{\frac{1}{\alpha}}}$ and leads to the following estimate on the total cost of the DLQMC algorithm,

$$\hat{C}_{dlqmc} \sim \mathcal{C}N + \mathcal{C}_*\frac{V^*}{\varepsilon^{\frac{1}{\alpha}}}, \tag{3.41}$$

with $\mathcal{C}, \mathcal{C}_*$ being the computational cost of a single CFD forward solve and evaluation of the deep neural network $\mathcal{L}^*$, respectively.

Dividing (3.41) with (3.38) and using $J_q \sim \frac{V^{\Delta}}{\varepsilon^{\frac{1}{\alpha}}}$ leads to the speedup estimate (3.40).

□

Thus, we are guaranteed a speedup in computing the probability distribution of the parameters to observable map as long as $V^* \sim V^{\Delta}$. Hence, we can employ the DLQMC algorithm to approximate the full probability distribution of the parameters to observable map efficiently. Moreover, the DLMC algorithm can also be similarly adapted to compute the probability distribution, more efficiently than the baseline MC algorithm.

4 Implementation

In this section, we provide some details on the concrete implementation of our deep learning algorithm 2.1 and the UQ algorithms 3.1 and 3.5.

4.1 Finite Volume method

In this paper, all our numerical experiments are performed for the compressible Euler equations. In two dimensions, these equations are

$$\mathbf{U}_t + \mathbf{F}^x(\mathbf{U})_x + \mathbf{F}^y(\mathbf{U})_y = 0, \quad \mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix}, \quad \mathbf{F}^x(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho v^2 + p \\ \rho u v \\ (E+p)v \end{pmatrix}, \quad \mathbf{F}^y(\mathbf{U}) = \begin{pmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ (E+p)v \end{pmatrix} \tag{4.1}$$

where $\rho, \mathbf{u} = (u, v)$ and p denote the fluid density, velocity and pressure, respectively. The quantity E represents the total energy per unit volume

$$E = \frac{1}{2}\rho|\mathbf{u}|^2 + \frac{p}{\gamma - 1},$$

where $\gamma = c_p/c_v$ is the ratio of specific heats, chosen as $\gamma = 1.4$ for our simulations. Additional important variables associated with the flow include the speed of sound $a = \sqrt{\gamma p/\rho}$, the Mach number $M = |\mathbf{u}|/a$ and the fluid temperature T evaluated using the ideal gas law $p = \rho R T$, where R is the ideal gas constant.

The solutions (4.1) are approximated by finite volume schemes [22], which are formulated by discretizing the computational domain $D \subset \mathbb{R}^2$ into a the union of non-overlapping control volumes $\boldsymbol{\Omega}_i$. Integrating the system (4.1) over the control volume $\boldsymbol{\Omega}_i$ and approximating the cell-boundary integrals of the flux using an N-point quadrature rule, leads to the following semi-discrete finite volume scheme

$$\frac{d\mathbf{U}_i}{dt} = -\frac{1}{|\boldsymbol{\Omega}_i|}\sum_{j=1}^{N} w_{ij}\mathbf{F}^{\Delta}_{ij} \tag{4.2}$$

where $\mathbf{U}_i(t)$ approximates the cell average in the cell $\boldsymbol{\Omega}_i$, $\mathbf{F}^{\Delta}_{ij} = \mathbf{F}^{\Delta}(\mathbf{U}^-_{ij}, \mathbf{U}^+_{ij}, \mathbf{n}_{ij})$, is the numerical flux at the cell-boundary quadrature point $\mathbf{x}_{ij}$ with outward normal $\mathbf{n}_{ij}$, that needs to satisfy,

- consistency, i.e, $\mathbf{F}^{\Delta}(\mathbf{U}, \mathbf{U}, \mathbf{n}) = \mathbf{F}^x(\mathbf{U})n^x + \mathbf{F}^y(\mathbf{U})n^y$,
- conservation, i.e, $\mathbf{F}^{\Delta}(\mathbf{U}_1, \mathbf{U}_2, \mathbf{n}) = -\mathbf{F}^{\Delta}(\mathbf{U}_1, \mathbf{U}_2, -\mathbf{n})$,

while w_{ij} is the quadrature weight. The numerical flux is evaluated using the interface values $\mathbf{U}^-_{ij}$ and $\mathbf{U}^+_{ij}$, which are obtained by suitably reconstructing the solution in the cell $\boldsymbol{\Omega}_i$ and the neighbouring cell sharing the quadrature point x_{ij}, respectively, with piecewise linear polynomials. Details of the reconstruction and the scheme can be checked from [40].

The system of ODEs (4.2) is integrated in time using a time-marching strategy, such as a Runge-Kutta scheme.

4.2 Implementation of ensemble training

All the following numerical experiments will use fully connected neural networks with ReLU activation functions. Essential hyperparameters are listed in table 2. For each experiment, we use a reference network architecture i.e number of hidden layers and width per layer, such that the total network size is $O(d + N)$. This ensures consistency with the prescriptions of lemma 2.5. The reference network architecture and its perturbations are described in each numerical experiment below.

We use two choices of the exponent p in the loss function (2.11) i.e either $p = 1$ or $p = 2$, denoted as mean absolute error (MAE) or mean square error (MSE), respectively. Similarly the exponent q in the regularization term (2.12) is either $q = 1$ or $q = 2$. In order to keep the ensemble size tractable, we chose four different values of the regularization parameter λ in (2.12) i.e, $\lambda = 7.8 \times 10^{-5}, 7.8 \times 10^{-6}, 7.8 \times 10^{-7}, 0,$

corresponding to different orders of magnitude for the regularization parameter. For the optimizer of the loss function, we use either a standard SGD algorithm or ADAM. Both algorithms are used in the *full batch* mode. This is reasonable as the number of training samples are rather low in our case.

A key hyperparameter is the starting value $\bar{\theta}$ in the SGD or ADAM algorithm (2.13). We remark that the loss function (2.12) is non-convex and we can only ensure that the gradient descent algorithms converge to a local minimum. Therefore, the choice of the initial guess is quite essential as different initial starting values might lead to convergence to local minima with very different loss profiles and generalization properties. Hence, it is customary in machine learning to *retrain* i.e, start with not one, but several starting values and run the SGD or ADAM algorithm in parallel. Then, one has to select one of the resulting trained networks. The ideal choice would be to select the network with the best generalization error. However, this quantity is not accessible with the data at hand. Hence, we rely on the following surrogates for predicting the best generalization error,

1. **Best trained network.** (*train*): We choose the network that leads to the lowest training error (2.31), once training has been terminated.

2. **Best validated network.** (*val*): We choose the network that leads to the lowest validation loss (2.32), once training has been terminated. One disadvantage of this approach is to sacrifice some training data for the validation set.

3. **Best trained network wrt mean.** (*mean-train*): We choose the network that has minimized the error in the mean of the training set i.e,

$$\mathcal{E}_{mean} := \left| \frac{1}{N} \sum_{j=1}^{N} \mathcal{L}^{\Delta}(y_j) - \frac{1}{N} \sum_{j=1}^{N} \mathcal{L}^{*}(y_j) \right|, \quad \forall y_j \in \mathcal{S}. \tag{4.3}$$

4. **Best trained network wrt Wasserstein.** (*wass-train*): We choose the network that has minimized the error in the Wasserstein metric with respect to the training set $\mathcal{S}$:

$$\mathcal{E}_{wass} := W_1 \left(\frac{1}{N} \sum_{j=1}^{N} \delta_{\mathcal{L}^{\Delta}(y_j)}, \frac{1}{N} \sum_{j=1}^{N} \delta_{\mathcal{L}^{*}(y_j)} \right), \quad \forall y_j \in \mathcal{S}. \tag{4.4}$$

We note that Wasserstein metric for measures on $\mathbb{R}$ can be very efficiently computed with the Hungarian algorithm [34].

All the above criteria can be computed with the available training (and validation) data. There are subtle differences between them and we hope to identify which one of them can pick out the network with smallest generalization (prediction) error.

Another hyperparameter (property) is whether we choose randomly distributed points (Monte Carlo) or more uniformly distributed points, such as quasi-Monte Carlo (QMC) quadrature points, to select the training set $\mathcal{S}$. We use both sets of points in the numerical experiments below in order to ascertain which choice is to be preferred in practice.

There are other hyperparameters in the machine learning algorithm, which in principle, could be varied within the ensemble training procedure. However, we have decided to keep them fixed for all runs, in order to keep the ensemble training procedure tractable. These include the *learning rate* in (2.13). We keep it fixed for all runs by setting $\eta_r = 0.01, \forall r$, for both the SGD and ADAM algorithms. Moreover, we also set the total number of epochs in both optimizers to 500000.

4.3 Code

All the numerical results presented below are based on the following codes,

- The training and test set for the one-dimensional Euler equations are generated using an efficient implementation of a finite-volume solver in MATLAB. A semi-discrete formulation is used, with a second-order reconstruction of the solution at the cell-interfaces using an ENO algorithm. Time marching is performed using a suitable Runge-Kutta method.

- The training and test set for the two-dimensional Euler equations are generated using TEnSUM, which is a vertex-centered finite-volume solver for compressible flows on unstructured triangular grids. The code has been implemented in C++, and parallelized using MPI standards. The unstructured meshes are generated using Gmsh, and partitioned using METIS. Further details can be found in [40] and Chapter 10 of [41].
- The machine learning and ensemble training runs are performed by a collection of Python scripts and Jupyter notebooks, utilizing Keras [54] and Tensorflow [55] for machine learning and deep neural networks. The ensemble runs over hyperparameters are parallelized as standalone processes, where each process trains the network for one choice of hyperparameters.

The training and test data are generated with the finite volume codes and are fed to the TensorFlow/KERAS machine learning package. Jupyter notebooks for all the numerical experiments can be downloaded from `https://github.com/kjetil-lye/learning_airfoils`, under the MIT license.. A lot of care is taken to ensure reproducability of the numerical results. All plots are labelled with the git commit SHA code, that produced the plot, in the upper left corner of the plot.

5 Numerical results

5.1 A stochastic shock tube problem

As a first numerical example, we consider a shock tube problem with the one-dimensional version of the compressible Euler equations (4.1).

5.1.1 Problem description

The initial conditions are prescribed by perturbing the initial profile for the shock tube test proposed by Sod [19], given by a left state (ρ_L, u_L, p_L) and a right state (ρ_R, u_R, p_R) at the initial discontinuity x_0. More specifically, we set

$$\begin{aligned} x_0(y) = \varepsilon G_1(y), \quad \rho_L(y) = 1 + \varepsilon G_2(y), \quad \rho_R(y) = 0.125(1 + \varepsilon G_3(y)), \\ u_L(y) = u_R(y) = \varepsilon^2 G_4(y), \quad p_L(y) = 1 + \varepsilon G_5(y), \quad p_R(y) = 0.1(1 + \varepsilon G_6(y)), \end{aligned} \tag{5.1}$$

where $\varepsilon = 0.1$, $y \in Y = [0,1]^6$ is a parameter and $G_k(y) = 2y_k - 1$ for $k = 1, ..., 6$.

The solution for each sample, at the final time $T_f = 1.5$ on the computational domain $[-5, 5]$ consists of a rarefaction wave, a contact discontinuity and a shock (see Figure 3(a)). The observables for this problem are chosen as the average integral of the density over fixed intervals

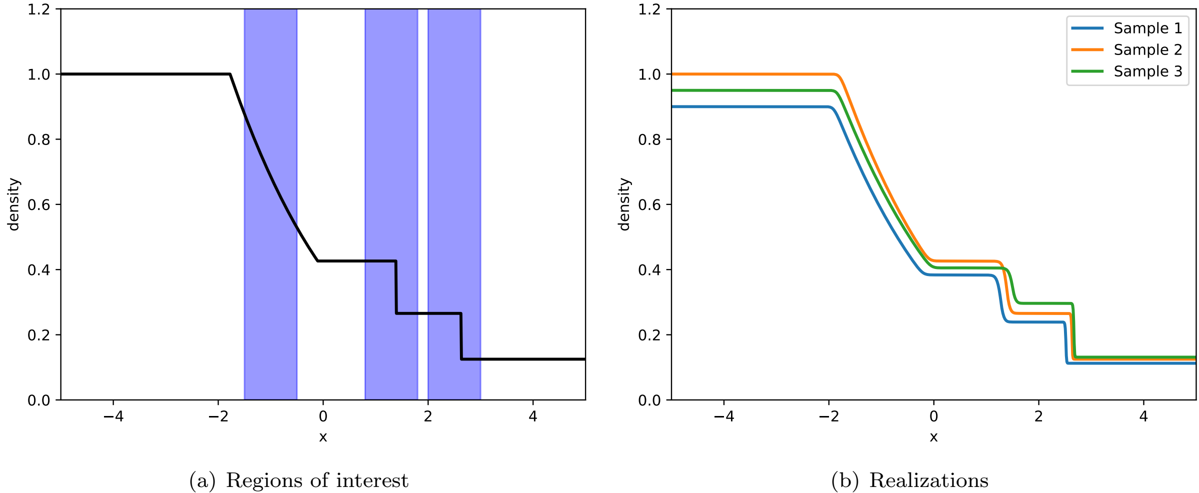
$$\mathcal{L}_j(y) = \frac{1}{|I_j|} \int_{I_j} \rho(x, T_f; y) dx, \quad j = 1, 2, 3, \tag{5.2}$$

where $I_1 = [-1.5, -0.5]$ marks the solution region inside the rarefaction fan, while $I_2 = [0.8, 1.8]$ and $I_3 = [2, 3]$ encompass the contact and shock discontinuities, respectively.

5.1.2 Generation of training data.

In order to generate the training, validation and test sets, we denote $\mathcal{Q}^{sob}$ as the first 2000 Sobol points on the domain $[0,1]^6$. For each point $y_j \in \mathcal{Q}^{sob}$, the maps or (rather their numerical surrogate) $\mathcal{L}^{\Delta}_{1,2,3}(y_j)$ is generated from high-resolution numerical approximations $\mathbf{U}^{\Delta}(y)$ obtained using a second-order finite volume scheme. The domain is discretized into a uniform mesh of $K = 1000$ disjoint intervals of uniform mesh size $\Delta = 10/K$. We choose the Lax-Friedrichs numerical flux, with the left and right cell-interface values obtained using a second-order ENO reconstruction. Time marching is performed using the fourth-order Runge-Kutta method with a CFL number of 0.5. Once the finite volume solution is computed, the corresponding observables (5.2) at each y are approximated as

$$\mathcal{L}^{\Delta}_j(y) = \frac{\sum_{i \in \Lambda_j} \rho_i(y)}{\#(\Lambda_j)}, \quad \Lambda_j = \{i \mid x_i \in I_j\}, \quad j = 1, 2, 3, \tag{5.3}$$



(a) Regions of interest

(b) Realizations

Figure 3: Solution (density) for the 1D Sod shock tube problem. a) Exact solution with $\varepsilon = 0$. The blue shaded zones mark the x-intervals over which observables are evaluated; b) Numerical solutions approximated with three different realizations of the parameter y.

Layer	**Width (Number of Neurons)**	**Number of parameters**
Hidden Layer 1	10	70
Hidden Layer 2	10	110
Hidden Layer 3	10	110
Hidden Layer 4	10	110
Hidden Layer 5	10	110
Output Layer	1	11
		521

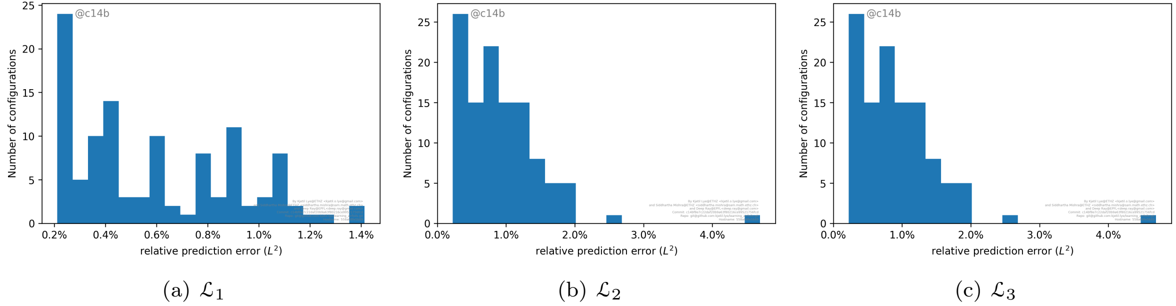
Table 3: Reference network architecture for all three networks approximating $\mathcal{L}_{1,2,3}$, in the Sod shock tube problem.

where x_i is the barycenter of the cell Ω_i. The numerical solutions for the three different realizations of the parameter y, are shown in Figure 3(b).

5.1.3 Results of the Ensemble training procedure.

As described in section 4.2, we set up an ensemble of neural networks by choosing different hyperparameters, listed in table 2. We fix a reference network architecture for this problem as a fully connected network with the number of hidden layers (and neurons per layer) listed in Table 3. Our first objective would be to train networks with this structure, with a training set $\mathcal{S} \subset \mathcal{Q}^{sob}$ with $\mathcal{S} = \{y_j\}, 1 \leqslant j \leqslant N$. We set $N = 128$. Thus, our network size is clearly consistent with the requirements of lemma 2.5. Moreover, we set $\mathcal{V} = \{y_j\}, N \leqslant j \leqslant 2N$ as the validation set. In the first instance, we vary all the hyperparameters as described in section 4.2 on this particular network structure. This generates an ensemble of 113 configurations (samples). Each sample is retrained 5 times by starting with different starting values for the optimization algorithm. Thus, we train a total of 565 networks for each observable, corresponding to different hyperparameter configurations, in parallel.

For each configuration, we select the retraining that minimizes the set selection criteria for the configuration i.e, one of *train*, *val*, *mean-train* or *wass-train*. Once the network with best retraining is selected, the resulting network is evaluated on the whole test set $\mathcal{T} = \mathcal{Q}^{sob}$ and the prediction error (2.27) is evaluated. We set $p = 2$ in (2.27) to calculate the root mean square prediction error. Moreover, we normalize to calculate the *percentage relative root mean square error*. The histogram, describing the probability



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 4: A histogram depicting distribution of the percentage relative mean L^2 prediction error (2.27) (X-axis) over number of hyperparameterization configurations (samples, Y-axis) for the three observables of the Sod Shock tube problem. Note that the range of X-axis is different in the plots.

distribution over all hyperparameter samples, of the percentage relative mean square prediction error for $\mathcal{L}^*_{1,2,3}$ is shown in figure 4. From this ensemble, we choose the hyperparameter configuration that leads to the smallest prediction error and list them in table 4. As seen from table 4, the best hyperparameter combination varies for the three different observables, at least in some respects. The same optimizer (ADAM) is used in all three cases and we only penalize (regularize) the L^2 norm of the weights. The choice of loss function differs for at least one of them, as does the selection criteria for the best retraining. Moreover, we also highlight a hyperparameter configuration that was nearly optimal in all three cases. This network is termed as an *effective network* and its hyperparameters are listed in table 5. This effective network architecture was also found to perform very well for the flow past airfoils test case, considered in section 5.2.

We plot the training (and validation) loss, with respect to number of iterations (epochs) of ADAM, for all three best performing networks and for the *effective network*, for the three different observables in figure 5. As seen from this figure, the training process reduces the loss function by four to five orders of magnitude over 500000 epochs of the ADAM minimization algorithm. Moreover, the validation error (2.32) also reduces by a similar amount during the training process.

The prediction errors with the best performing and effective networks, are depicted in figure 6, in the form of a scatter plot for $\mathcal{L}^{\Delta}_i(y)$ (X-axis) vs $\mathcal{L}^*_i(y)$ (Y-axis) for all $y \in \mathcal{T}$ and $i = 1,2,3$. The spread away from the diagonal visualizes the error in predictions. As shown in this figure, the errors for all three observables (with both the best performing and effective networks) are very low. This is further verified from table 4, from which we observe that the mean prediction errors with best performing (and effective) networks are always less than 0.3%. Moreover, the standard deviation in the error is several orders of magnitude lower than the mean error. The attainment of such a low prediction error is really impressive given that our training set had only $N = 128$ samples. Theoretical considerations in section 2.3.3 estimated the generalization error by $\sqrt{\frac{M}{N}}$, with M, N being the number of parameters in the network and number of training samples respectively. Setting these numbers for the problem here, yields a relative error of approximately 200%. However, our prediction errors are around 1000 times (three orders of magnitude) less, indicating the very high degree of compression that the trained neural network is able to achieve. This is really impressive when compared to the simple sum of sines example (see Table 1) as we obtain errors that are at least two to three orders of magnitude lower.

The computational costs of generating a single sample of training data, the average cost of training each network (over hyperparameter configurations) and the cost of evaluating the trained networks are shown in table 6. As seen from the table, the cost of evaluating a single neural network in this case is *five orders of magnitude* smaller than the cost of generating a single training sample. On the other hand, the training cost per network configuration is relatively high, but still comparable to the total cost of generating all the 128 training samples. Furthermore, it should be emphasized that attaining a relative error of less than 0.3% in predicting the observables, costs five orders of magnitude less (see table 6) when compared to computing the observables with a high-resolution finite volume method. *This illustrates the potential of neural networks – a well-trained network can solve the problem to almost the*

Obs	Opt	Loss	L^1-reg	L^2-reg	Selection	*BP.* Err mean (std)	Ref. Err mean (std)
$\mathcal{L}_1$	ADAM	MSE	0.0	7.8×10^{-6}	wass-train	0.211 (1.3×10^{-3})	0.223 (1.7×10^{-3})
$\mathcal{L}_2$	ADAM	MSE	0.0	7.8×10^{-7}	mean-train	0.234 (9.2×10^{-4})	0.234 (5.4×10^{-4})
$\mathcal{L}_3$	ADAM	MAE	0.0	7.8×10^{-5}	val	0.273 (3.2×10^{-4})	0.285 (6.6×10^{-4})

Table 4: The hyperparameter configurations that correspond to the best performing network (one with least mean prediction error) for the three observables of the Sod Shock tube problem. The mean and standard deviation of the relative percentage L^2-prediction error for the best performing networks are compared with the corresponding errors (denoted by Ref. Err) of the *Effective network*, with hyperparameters listed in table 5

Opt	Loss	L^1-reg	L^2-reg	Selection
ADAM	MSE	0.0	7.8×10^{-7}	wass-train

Table 5: Hyperparameters of the *effective network*. Opt refers to optimizer. Loss function to (2.11), $L^{1,2}$-reg to the parameter λ in (2.12) and selection to the criteria, among the four listed in section 4.2, by which we select the best retraining.

same accuracy as a high-resolution numerical method, essentially at zero cost (when compared to the cost of the underlying numerical method).

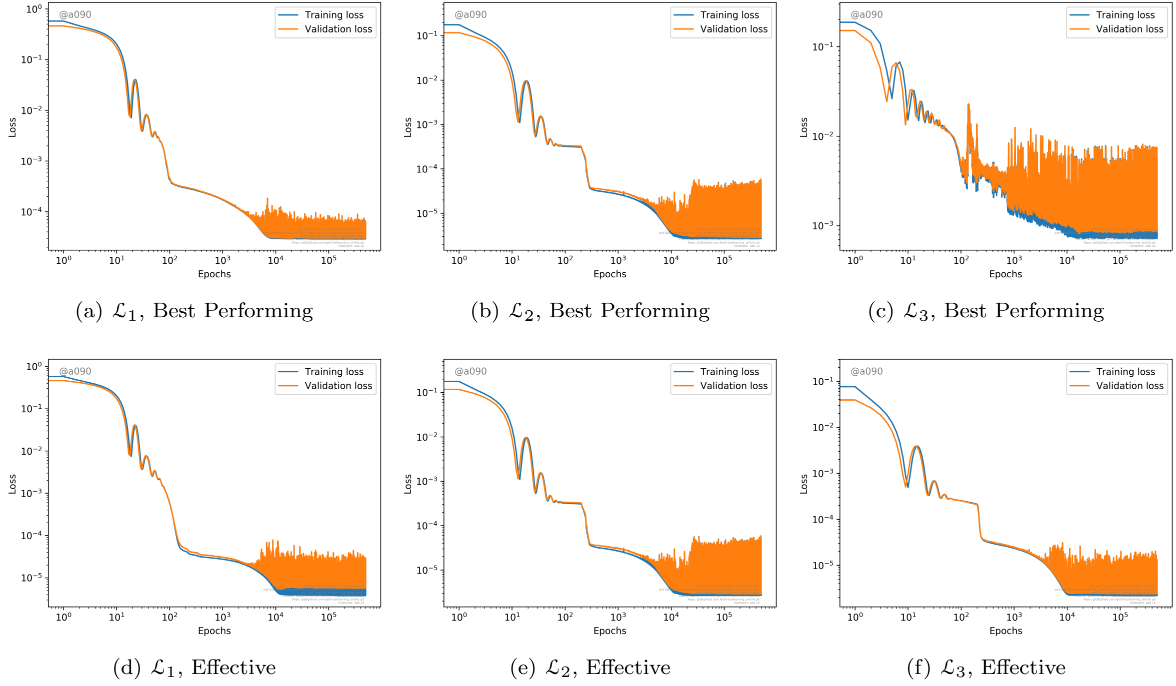
5.1.4 Network sensitivity to hyperparameters.

The ensemble training procedure allows to determine how sensitive the performance (measured in terms of prediction error) of the network is to the choice of relevant hyperparameters. We summarize the results of this sensitivity study below,

- *Overall sensitivity.* The overall sensitivity to hyperparameters is best depicted in the form of histograms, representing the distribution, over samples (hyperparameter configurations), shown in figure 4. As seen from the figure, the prediction errors with different hyperparameters are spread over a range for each observable. There are a few outliers which perform very poorly, particularly for the observables $\mathcal{L}_2, \mathcal{L}_3$, for which this range is more than an order of magnitude in terms of the prediction error . On the other hand, a large proportion of the 113 hyperparameter configurations (samples) concentrate around the best performing network. Our aim is to identify these configurations below.
- *Choice of optimizer.* The difference between ADAM and the standard SGD algorithm is shown in figure 7. As seen from this figure, ADAM clearly outperforms the standard SGD algorithm consistently in terms of the best performing networks. On the other hand, there are a few outliers

	Time (in secs)
Sample generation	3.0
Training ($\mathcal{L}_1$)	570
Evaluation ($\mathcal{L}_1$)	6×10^{-6}
Training ($\mathcal{L}_2$)	640
Evaluation ($\mathcal{L}_2$)	9×10^{-6}
Training ($\mathcal{L}_3$)	700
Evaluation ($\mathcal{L}_3$)	10^{-5}

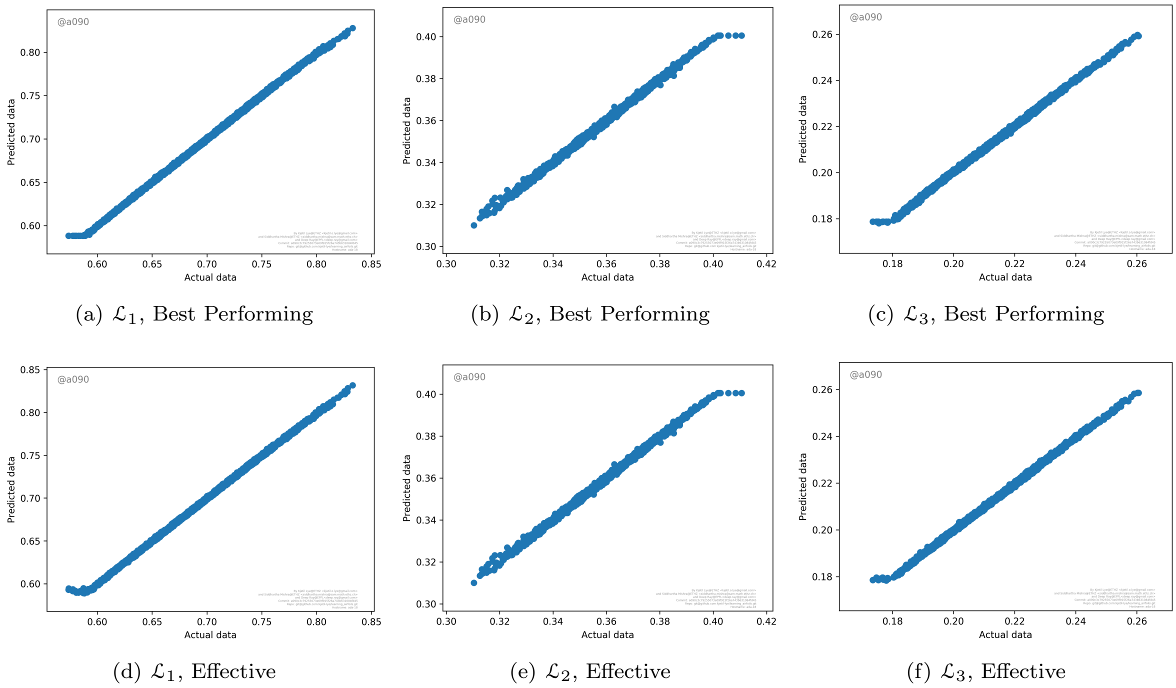
Table 6: Computational times (cost) of (single) sample generation, network training and (single) network evaluation, for the Sod shock tube problem. The samples were generated on an Intel(R) Core(TM) i5 @ 3.1GHz machine. The training and evaluation of the neural networks were all performed on a Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz machine. The training and evaluation times are approximations of the average runtimes over all hyperparameter configurations.



(a) $\mathcal{L}_1$, Best Performing (b) $\mathcal{L}_2$, Best Performing (c) $\mathcal{L}_3$, Best Performing

(d) $\mathcal{L}_1$, Effective (e) $\mathcal{L}_2$, Effective (f) $\mathcal{L}_3$, Effective

Figure 5: The evolution over epochs of the training and validation loss functions for all the three functionals (5.2) of the Sod shock tube problem with the best performing (see Table 4) and effective (see Table 5) neural networks.



(a) $\mathcal{L}_1$, Best Performing (b) $\mathcal{L}_2$, Best Performing (c) $\mathcal{L}_3$, Best Performing

(d) $\mathcal{L}_1$, Effective (e) $\mathcal{L}_2$, Effective (f) $\mathcal{L}_3$, Effective

Figure 6: Scatter plots depicting prediction errors for all the three functionals of the Sod shock tube problem. X-axis (Test data), Y-axis: Predictions by best performing or effective neural networks.

within ADAM that seem to perform poorly. We call these outliers as the *bad set of ADAM* and they correspond to the configurations with MSE loss function and L^1 regularization, or L^2 regularization, with a regularization parameter $\lambda > 10^{-5}$, or MAE loss function without any regularization. The difference between the bad set of ADAM and its complement i.e, the good set of ADAM is depicted in figure 8. We see that the outliers (in prediction error) for observables $\mathcal{L}_2$ and $\mathcal{L}_3$ clearly correspond to the *bad set of ADAM.*

- *Choice of loss function.* The difference between L^1 mean absolute error (MAE) and L^2 root mean square error (MSE) i.e, exponent $p = 1$ or $p = 2$ in the loss function (2.11) is plotted in the form of a histogram over the error distributions in figure 9. We only focus on the difference between MAE and MSE for the good set of ADAM and observe that the differences are rather minor. All tested configurations yield an error of less than one percent. Hence, the network performance is not that sensitive to the choice of loss function.

- *Choice of type of regularization.* This refers to whether we choose $q = 1$ or $q = 2$ in the regularization term in (2.12). We plot the corresponding histograms for prediction error distributions in figure 10. Again, we focus only on the good set of ADAM and observe from this figure that the choice of type of regularization leads to small differences in the overall prediction error. It appears that using an L^2 regularization is slightly better but all the tested configurations provide an error of less than one percent.

- *Value of regularization parameter.* The variation of network performance with respect to the value of the regularization parameter λ in (2.12) is shown in figure 11. In this figure, we vary the regularization parameter over four values namely $\lambda = 0, 7.8\times10^{-7}, 7.8\times10^{-6}, 7.8\times10^{-5}$ and plot the minimum and average (over all Good ADAM configurations) of relative percentage prediction error (Y-axis) with respect to λ (X-axis). We see from this figure that a small amount of regularization always outperforms no regularization. However, the differences are rather minor between different values of the regularization parameter.

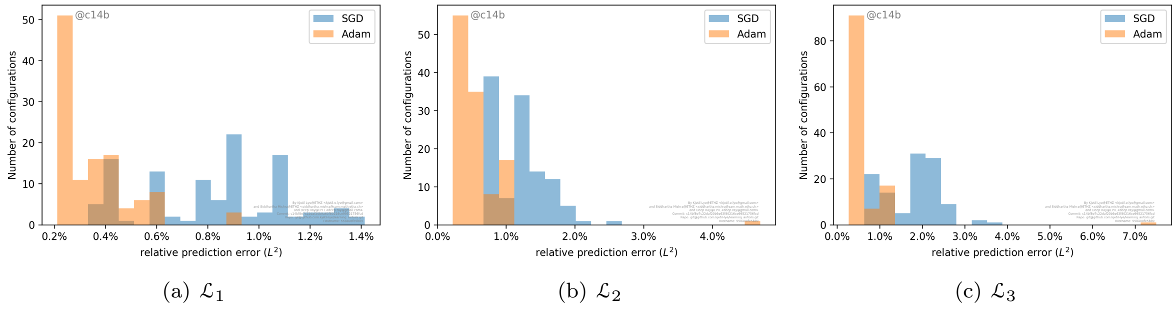
- *Choice of selection criteria.* The difference in network performance with respect to the four possible selection criteria, listed in section 4.2, is presented in figure 12. As seen from this figure, there is virtually no difference in performance between *wass-train* and *mean-train* and although the differences are minor, these two selection criteria perform slightly better than the standard selection criteria of *train* and *val*.

- *Sensitivity to retrainings.* This information is provided in table 7. In this table, we list the minimum, maximum, mean and standard deviation of the relative prediction error, over 5 retrainings, for the best performing networks and the effective network for each observable. As seen from this table, while there is some sensitivity to retrainings, the standard deviation is rather low in all cases, except for the best performing network approximating $\mathcal{L}_3$, where the standard deviation is comparable to the mean over the retrainings. However, for the effective network, the sensitivity to retrainings seems to be very low in all cases. Hence, one can expect that starting the optimization algorithm from a random starting value for this network will generically lead to very low prediction error (less than 0.5%) for all observables of the Sod shock tube problem.

- *Variation of Network size.* In order to ascertain the dependence of network performance on the size of the network, we only consider the hyperparameter configuration that corresponds to the effective network (table 5). For this configuration, we consider a matrix of network sizes by varying the depth (number of hidden layers) for three values of $4, 8, 16$ and the width (number of neurons per layer) for three values of $6, 12, 24$. Note that even the smallest network (width of 6 and depth of 4) has 174 tunable parameters and satisfies the requirements of lemma 2.5. Each network is retrained 5 times and the best performing one is chosen based on the *wass-train* criteria. The percentage relative mean prediction error for the best network for each value of this 3×3 matrix of all three observables is shown in table 8. We observe from the table that the error was very similar for most values in the matrix. In fact, there was no advantage in increasing the depth or the width. On the other hand, the training procedure failed for some very deep and very wide networks. This failure

Obs (Network)	Err (min)	Err (max)	Err (mean)	Err (std)
$\mathcal{L}_1$ (Best Performing)	0.211	0.471	0.311	0.112
$\mathcal{L}_1$ (Effective)	0.223	0.263	0.235	0.015
$\mathcal{L}_2$ (Best Performing)	0.234	0.422	0.308	0.076
$\mathcal{L}_2$ (Effective)	0.234	0.422	0.308	0.076
$\mathcal{L}_3$ (Best Performing)	0.273	0.912	0.552	0.264
$\mathcal{L}_3$ (Effective)	0.285	0.521	0.364	0.089

Table 7: Sensitivity of the best performing networks (listed in table 4) and the effective network (listed in table 5) to retrainings i.e starting values for ADAM, for the three observables in the Sod Shock tube. All errors are relative mean L^2 prediction error (2.27) in percentage and we list the minimum, maximum, mean and standard deviation of the error over 5 retrainings.



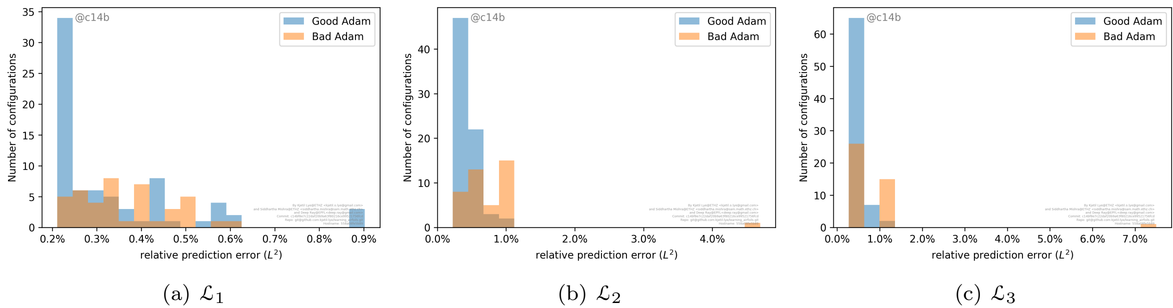
(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 7: Histograms for the prediction error, comparing ADAM with SGD, for the Sod shock tube problem.

was manifested in the output of NaNs, when evaluating the trained networks. We attribute this failure to the lack of sufficient number of samples for training these large networks.

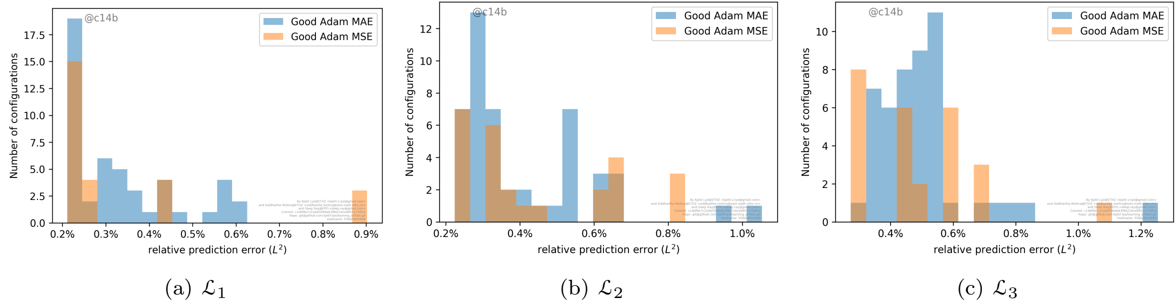
5.1.5 Network performance with respect to number of training samples.

A very important hyperparameter in deep learning is the size of the training set i.e, $N = \#(\mathcal{S})$. All the above results were obtained with $N = 128$. We have repeated the entire ensemble training with different samples sizes i.e $N = 32, 64, 256$ also and present a summary of the results in figure 13. In this figure, we plot the mean, minimum and maximum (over a subset of hyperparameter configurations) of the percentage relative L^2 prediction error, with respect to the number of samples, This subset consists of only ADAM as the optimizer, while varying the type of loss function, type of regularization, the value of regularization parameter and the selection criteria. Moreover, we only consider errors with selected



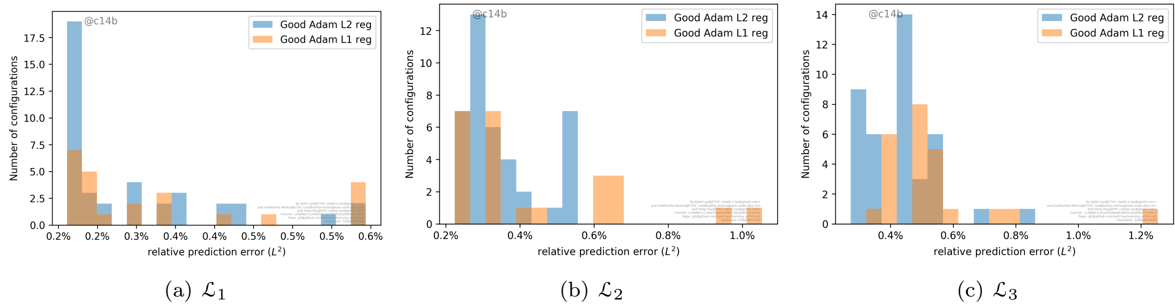
(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 8: Histograms for the prediction error, comparing the *good* ADAM with *bad* ADAM for the Sod shock tube problem



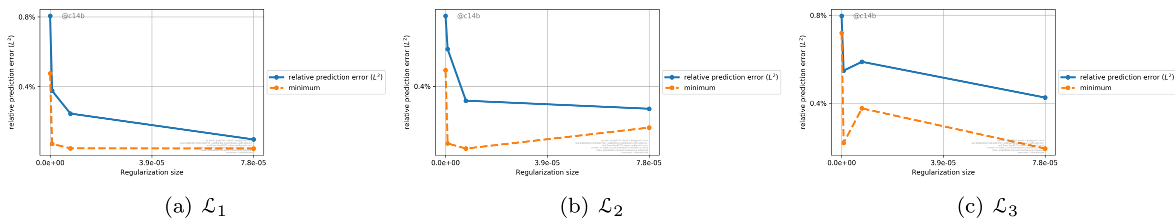
(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 9: Histograms for the prediction error, comparing the mean absolute error (MAE) and mean square error (MSE) as loss functions, for the Sod shock tube problem.



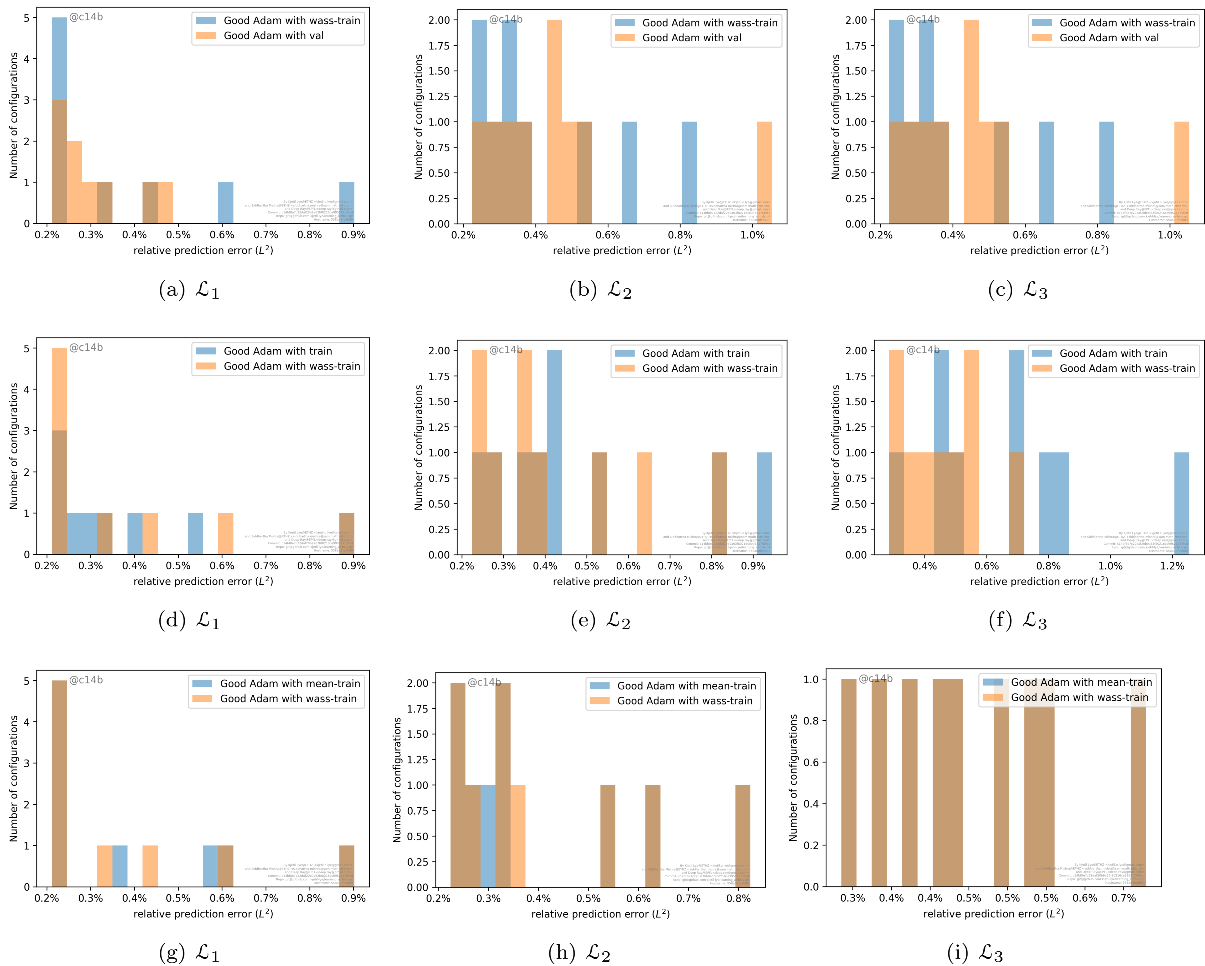
(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 10: Histograms for the prediction error, comparing ADAM with L^1 and L^2 regularization, for the Sod Shock tube problem.)



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 11: Variation of prediction error (Y-axis) with respect to the size of the regularization parameter λ in (2.12) (X-axis). The minimum and mean of prediction error (over all hyperparameter configurations) is shown.



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

(d) $\mathcal{L}_1$ (e) $\mathcal{L}_2$ (f) $\mathcal{L}_3$

(g) $\mathcal{L}_1$ (h) $\mathcal{L}_2$ (i) $\mathcal{L}_3$

Figure 12: Network performance with respect to selection criteria for retrainings (see section 4.2 i.e, *train,val,mean-train,wass-train*) for the Sod shock tube problem. We compare *wass-train* with each of the other three criteria. Histrograms for prediction error (X-axis) with number of configurations (Y-axis) are shown.

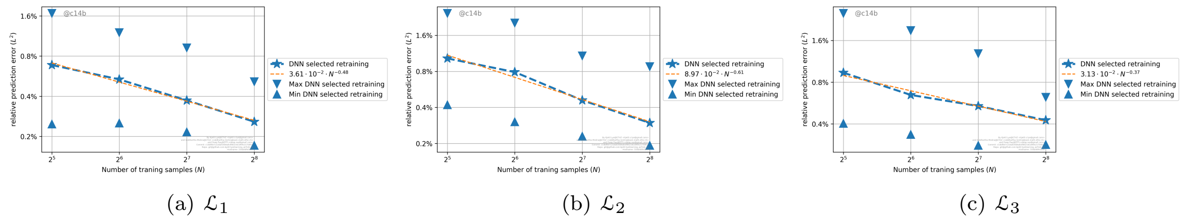
(optimal) retrainings in each case. As seen from the figure, the prediction error *decreases with sample size* N as predicted by the theory. The rate of decay is estimated to be 0.48, 0.61 and 0.37 for $\mathcal{L}_{1,2,3}$, respectively. This decay is very close to the theoretical predicted value, see (2.28). It is instructive to consider the observable $\mathcal{L}_1$ as in this case, the predication error scales very close to the theoretical value of 0.5 in (2.28). The constant in (2.28) is calculated from the best fit in figure 13 as $U = 0.036$. This is four orders of magnitude lower than the pessimistic upper bound provided by the number of tunable parameters in the network. This indicates very high compressibility in this problem [2] and allows us to obtain very low prediction errors with few training samples.

5.1.6 UQ with deep learning

In this section, we will approximate the underlying probability distributions (measures) (3.33), with respect to each of the observables $\mathcal{L}_{1,2,3}$ for the stochastic Sod shock tube problem. A reference measure is computed from the whole test set of 2000 samples and the corresponding histograms are shown in figure 14 to visualize the probability distributions. We approximate these distributions with the DLQMC algorithm 3.5. The corresponding histograms, computed with the best performing networks for each observable (listed in table 4) and the effective network (see table 5) are also shown in figure 14. The

Width/Depth	6	12	24
4	[0.23, 0.23, 0.26]	[0.24, 0.25, 0.30]	[0.24, 0.29, 0.25]
8	[0.27, 0.24, 0.28]	[0.21, 0.20, 0.27]	[0.25, 0.29, 0.31]
16	[0.49, 0.28, 0.60]	[¬, ¬, 0.34]	[¬, 0.20, 0.88]

Table 8: Performance of trained neural networks with respect to the variation of network size for Sod shock tube problem. The rows represent variation with respect to Width (number of neurons per hidden layer) and the columns represent variation with respect to Depth (number of hidden layers). For each entry of the Width-Depth matrix, we tabulate the vector of relative percentage mean prediction error for the effective network (see table 5). The components of each vector represents the error in approximating $[\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3]$ and ¬ is used to indicate that the training procedure failed for this particular configuration.



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 13: Relative percentage mean square prediction error (2.27) (Y-axis) for the neural networks approximating the Sod shock tube problem, with respect to number of training samples (X-axis). For each number of training samples, we plot the mean, minimum and maximum (over hyperparameter configurations) of the prediction error. Only the selected (optimal) retraining is shown.

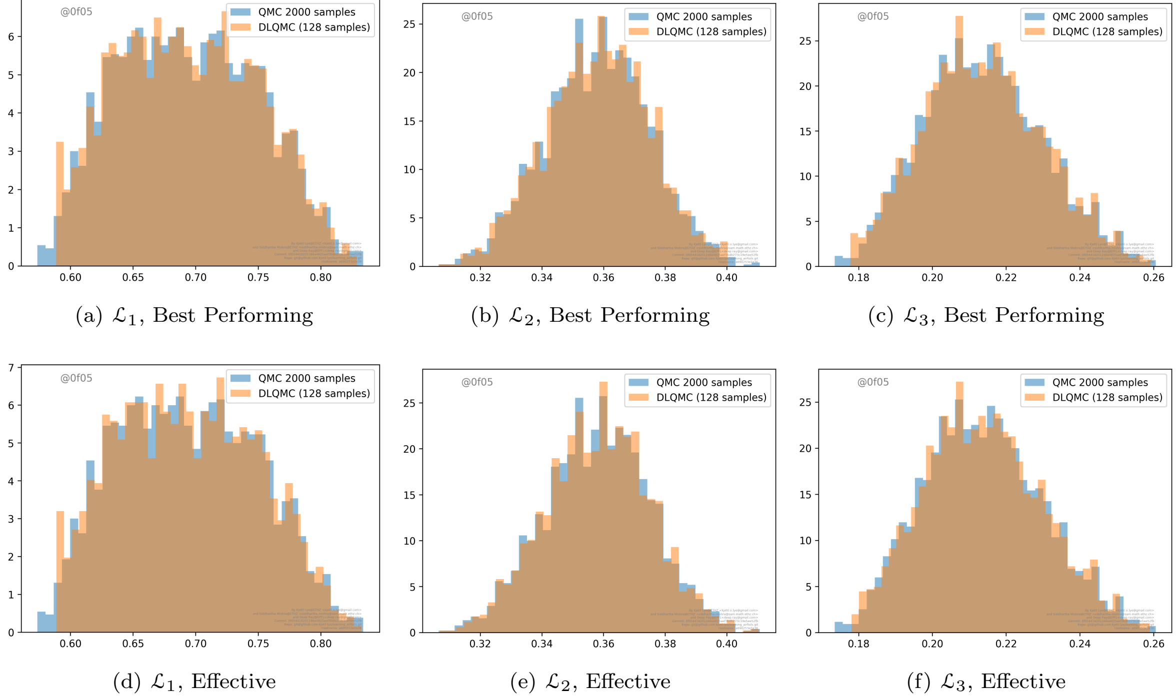
figures show that the histograms computed with the neural networks match the reference histograms to very high accuracy. Hence for all the three observables, there is a very good qualitative agreement between the outputs of the DLQMC algorithm and the reference measure.

In order to quantify the gain in computational efficiency with the DLQMC algorithm over the baseline QMC algorithm in approximating probability distributions, we will compute the *speedup*. To this end, we observe from figure 15 that the baseline QMC algorithm converges to the reference probability measure with respect to the Wasserstein distance (3.37), at a rate of $\alpha_i = 0.78, 0.73, 0.7$ in the number of QMC points, for the observables $\mathcal{L}_i$, with $i = 1, 2, 3$. Next, from table 6, we see that once the training samples have been generated, the cost of performing evaluations with the trained network is essentially free. Here, we are neglecting the training cost, which is non-trivial with respect to the sample generation costs for this particular problem. Consequently, if the number of training samples is N, we compute a *raw speed-up* (for each observable) given by,

$$\sigma_{i,dlqmc}^{raw} := \frac{W_1\left(\hat{\mu}_{i,qmc,N}, \hat{\mu}_i{}^{\Delta}\right)}{W_1\left(\hat{\mu}_{i,qmc}^{*}, \hat{\mu}_i{}^{\Delta}\right)}, \quad i = 1, 2, 3. \tag{5.4}$$

Observable	Network	Speedup
$\mathcal{L}_1$	Best Performing	16.81
$\mathcal{L}_1$	Effective	16.69
$\mathcal{L}_2$	Best Performing	20.80
$\mathcal{L}_2$	Effective	20.40
$\mathcal{L}_3$	Best Performing	7.39
$\mathcal{L}_3$	Effective	6.75

Table 9: Real speedups (5.5), for each observable of the Sod shock tube problem, comparing DLQMC with baseline QMC algorithm. Speedups for both best performing (table 4) and effective (table 5) networks are shown.



(a) $\mathcal{L}_1$, Best Performing (b) $\mathcal{L}_2$, Best Performing (c) $\mathcal{L}_3$, Best Performing

(d) $\mathcal{L}_1$, Effective (e) $\mathcal{L}_2$, Effective (f) $\mathcal{L}_3$, Effective

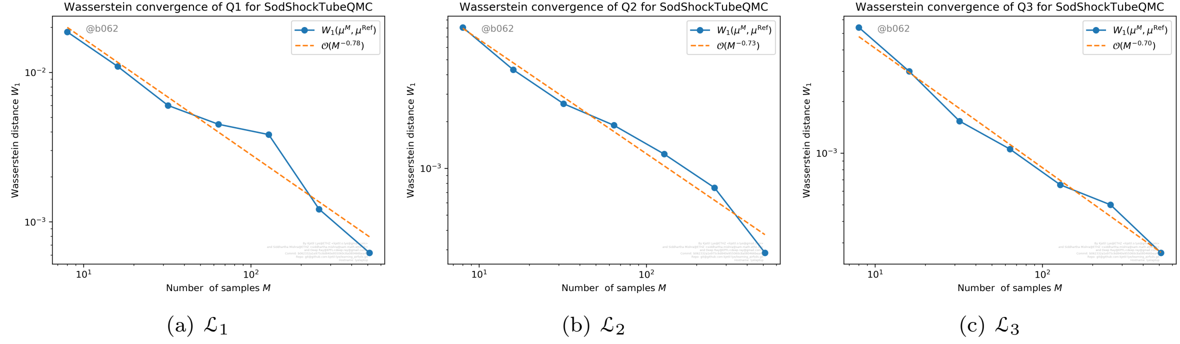
Figure 14: Empirical histograms representing the probability distribution (measure) for each observable of the Sod shock tube problem. We compare the reference histograms (computed with the test set $\mathcal{T}$) and the histograms computed with the DLQMC algorithm.

The observables $\mathcal{L}_{1,2,3}$ are indexed by i, $\hat{\mu}_i^{\Delta}$ is the reference measure (probability distribution) computed from the test set $\mathcal{T}$, $\hat{\mu}_{i,qmc,N}$ is the measure (3.34) with the baseline QMC algorithm and N QMC points, and $\hat{\mu}_{i,qmc}^{*}$ is the measure computed with the DLQMC algorithm (3.39) and $J_L = \#(\mathcal{T}) = 2000$ points in this case. The real speed up is then given by,

$$\sigma_{i,dlqmc}^{real} = \left(\sigma_{i,dlqmc}^{raw}\right)^{\frac{1}{\alpha_i}}, \quad i = 1, 2, 3, \tag{5.5}$$

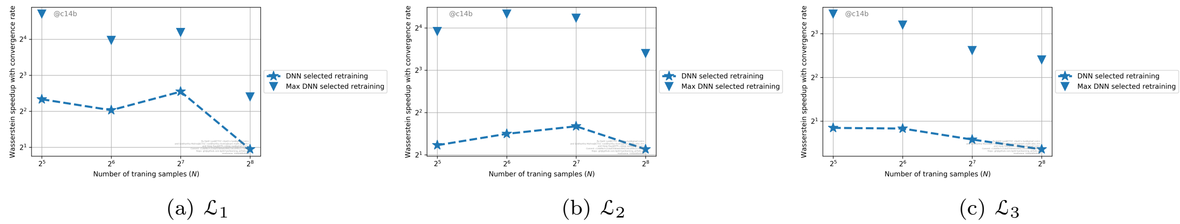
in order to compensate for the rate of convergence of the baseline QMC algorithm. In other words, our *real speed up* compares the costs of computing errors in the Wasserstein metric, that are of the same magnitude, with the DLQMC algorithm vis a vis the baseline QMC algorithm. Henceforth, we denote σ_{dlqmc}^{real} as the *speedup* and list the speedups (for each observable) with the DLQMC algorithm for $N = 128$ training samples in table 9. This table shows impressive speedups with the DLQMC algorithm for computing the underlying probability distributions, with both the best performing networks (table 4) and the effective network (table 5) ranging from approximately 7 (for the observable $\mathcal{L}_3$) to approximately 20 for the other two observables. This difference cannot just be explained by the prediction error for each observable (see table 4). It is indeed true that the prediction errors are slightly larger for $\mathcal{L}_3$ compared to the other two observables. However, this difference is too small to explain a factor of 3 difference in speedup. Rather, appealing to the estimate (3.40), we think that the *variation* associated with this particular functional is possibly smaller than the other two. Hence, the baseline QMC method performs well even with 128 samples. Consequently, the speedup is smaller but still very significant.

Finally, in figure 16, we plot the speedup of the DLQMC algorithm (over the baseline QMC algorithm) as a function of the number of training samples in the deep learning algorithm 2.1. As observed from this figure, the highest speedups are obtained when the number of training samples is 32 for observables $\mathcal{L}_{1,3}$ and 64 for the observable $\mathcal{L}_2$. Moreover, the speed up reduces when the number of samples is 256. This nonlinear variation can be explained in terms of the subtle competition between reducing the mean prediction error (which decays with increasing number of training samples, see figure 13) and how well is



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 15: Convergence of the Wasserstein distance $W_1\left(\hat{\mu}_{i,qmc}, \hat{\mu}_i{}^{\Delta}\right)$ (Y-axis) for each observable $\mathcal{L}_i$ of the Sod shock tube problem, with respect to number of QMC points (X-axis)



(a) $\mathcal{L}_1$ (b) $\mathcal{L}_2$ (c) $\mathcal{L}_3$

Figure 16: Real speedups (5.5) for the DLQMC algorithm over the baseline QMC algorithm (Y-axis) with respect to number of training samples (X-axis). The maximum and mean speed up (over all hyper-parameter configurations) are shown.

the reference measure approximated by the baseline QMC algorithm (which also decays with the number of training samples, see figure 15). It turns out that for this particular problem, a low number of training samples provides the best speedup.

Remark 5.1. All the theoretical results for speedup of the DLQMC algorithm over the baseline QMC algorithm, presented in section 3, assumed that the computational cost of training the deep neural networks is negligible when compared to the cost of generating the training data. The real speedups presented in table 9 reflect this assumption. However for the Sod shock tube problem, this assumption is not valid. As seen from table 6, the cost of generating $N = 128$ training samples is approximately 6 minutes and the cost of training the network is approximately $10 - 11$ minutes. Hence, we have to account for this training cost in our calculation of the speedup. To this end, we define a *compensated speed up* as follows: we compute the total cost of the DLQMC algorithm by adding the training and sample generation costs (the sample evaluation costs are infinitesimally small as seen from table 6). Consequently, the number of QMC samples which can be computed with this cost are calculated and the resulting error in Wasserstein distance W_1 with respect to the reference measure (computed with all the QMC samples in the test set) are estimated from the rate of convergence, calculated from figure 15. Then, the resulting error is divided by the error with the DLQMC algorithm and a compensated speedup is calculated by raising this ratio to the power $\frac{1}{\alpha}$ as in (5.5), with α being the rate of convergence i.e, $\alpha = \{0.78, 0.73, 0.7\}$ for $\mathcal{L}_{1,2,3}$, respectively. The compensated speedups with both the best performing and effective networks are shown in table 10. As seen from the table, the speedup drops when compared to the real speedups from table 9. However, they are still around an order of magnitude for $\mathcal{L}_{1,2}$ while being a factor or 2.5 for $\mathcal{L}_3$. ■

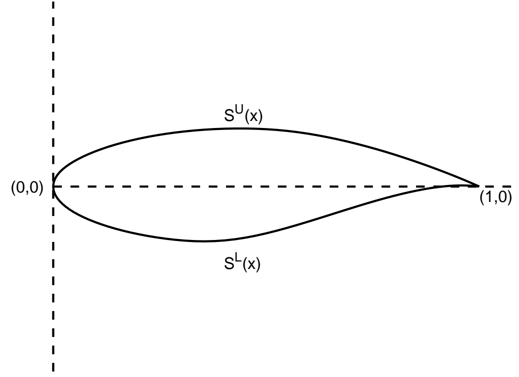


Figure 17: The shape of the RAE 2822 airfoil.

Observable	Network	Compensated Speedup
$\mathcal{L}_1$	Best Performing	6.81
$\mathcal{L}_1$	Effective	6.73
$\mathcal{L}_2$	Best Performing	7.78
$\mathcal{L}_2$	Effective	7.65
$\mathcal{L}_3$	Best Performing	2.62
$\mathcal{L}_3$	Effective	2.39

Table 10: Compensated speedups, calculated as described in Remark 5.1, that take into account training costs, for each observable of the Sod shock tube problem, comparing DLQMC with baseline QMC algorithm. Speedups for both best performing (table 4) and effective (table 5) networks are shown.

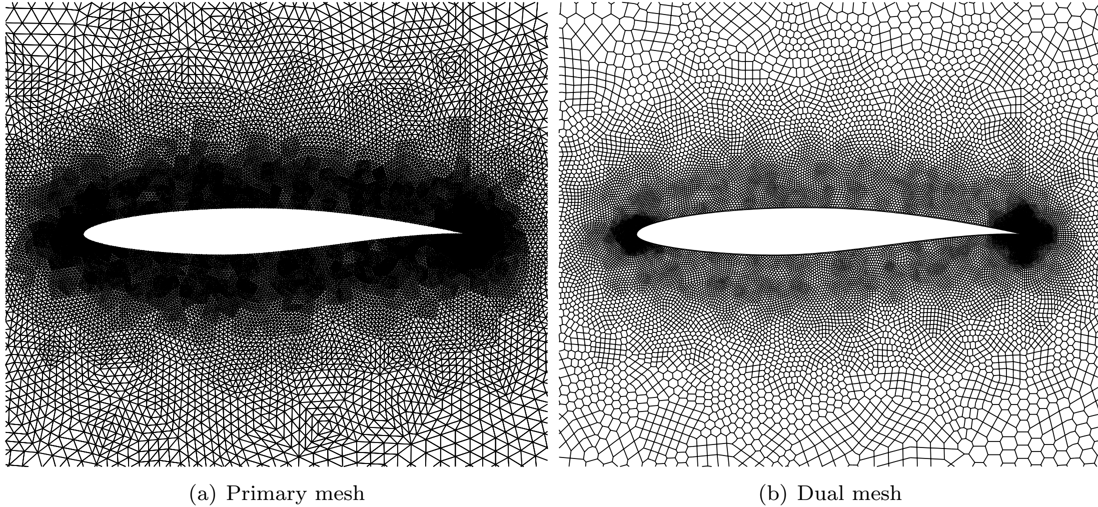


Figure 18: The primary and voronoi dual meshes for the RAE airfoil.

5.2 Flow past an airfoil

As a second example, we consider a more realistic problem involving the compressible Euler equations in two space dimensions (4.1).

5.2.1 Problem description.

We consider flow past an RAE 2822 airfoil, whose shape is shown in Figure 17. The flow past this airfoil is a benchmark problem for UQ in aerodynamics [23]. The upper and lower surface of the airfoil are denoted at $(x, S^U(x))$ and $(x, S^L(x))$, with $x \in [0,1]$. We consider the following parametrized (free-stream) initial conditions and perturbed airfoil shapes

$$\begin{aligned} &T^\infty(y) = 1 + \varepsilon_1 G_1(y), \quad M^\infty(y) = 0.729(1 + \varepsilon_1 G_2(y)), \quad \alpha(y) = [2.31(1 + G_3(y))]^\circ, \\ &p^\infty(y) = 1 + \varepsilon_1 G_4(y), \quad \hat{S}^U(x;y) = S^U(x)(1 + \varepsilon_2 G_5(y)), \quad \hat{S}^L(x;y) = S^L(x)(1 + \varepsilon_2 G_6(y)), \end{aligned} \tag{5.6}$$

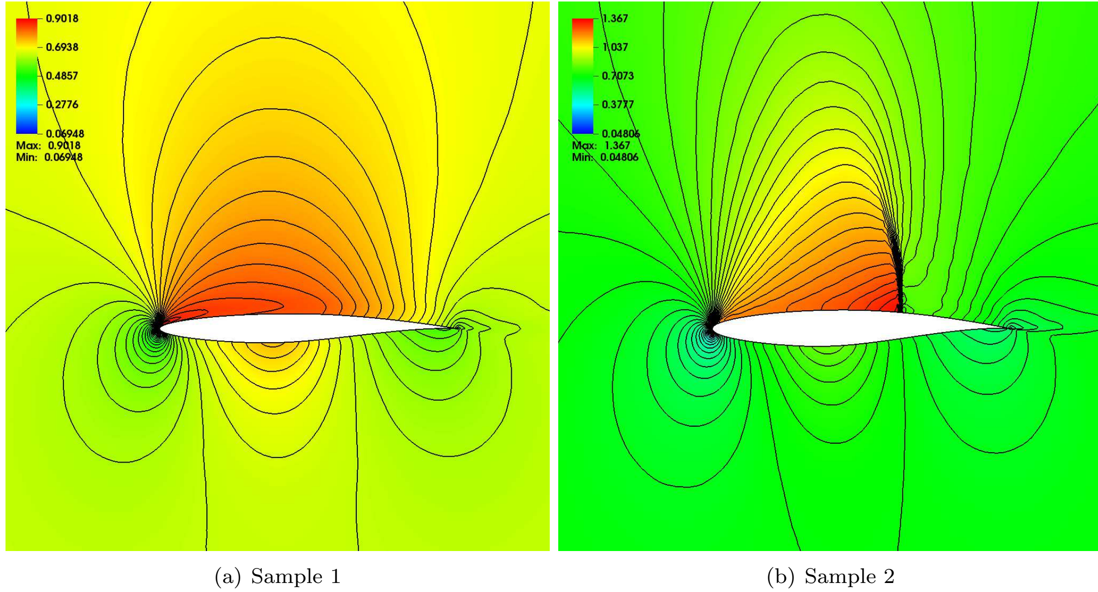
where α is the angle of attack, the gas constant $R = 1$, $\varepsilon_1 = 0.1$, $\varepsilon_2 = 0.2$, while y, G_k are the same as those used in (5.1).

The total drag and lift experienced on the airfoil surface $\hat{S}(y) = \hat{S}^L(y) \bigcup \hat{S}^U(y)$ are chosen as the observables for this experiment. These are expressed as

$$\text{Lift} = \mathcal{L}_1(y) = \frac{1}{\mathcal{K}(y)} \int_{\hat{S}(y)} p(y)(\mathbf{n}(y) \cdot \psi_l(y)) ds, \quad \text{Drag} = \mathcal{L}_2(y) = \frac{1}{\mathcal{K}(y)} \int_{\hat{S}(y)} p(y)(\mathbf{n}(y) \cdot \psi_d(y)) ds, \tag{5.7}$$

where $\mathcal{K}(y) = \rho^\infty(y)|\mathbf{u}^\infty(y)|^2/2$ is the free-stream kinetic energy, while $\psi_l(y) = [-\sin(\alpha(y)),\ \cos(\alpha(y))]$ and $\psi_d(y) = [\cos(\alpha(y)),\ \sin(\alpha(y))]$.

The domain is discretized using triangles to form the primary mesh, while the control volumes are the voronoi dual cells formed by joining the circumcenters of the triangles to the face mid-points. A zoomed view of the primary and dual meshes are shown in Figure 18. The solutions are approximated using a second-order finite volume scheme, with the mid-point rule used as the quadrature to approximate the cell-boundary integrals. The numerical flux is chosen as the kinetic energy preserving entropy stable scheme [40], with the solutions reconstructed at the cell-interfaces using a linear reconstruction with the minmod limiter . For details on flux expression and the reconstruction procedure, we refer interested readers to [40]. The scheme is integrated using the SSP-RK3 time integrator. The contour plots for the Mach number with two different realizations of the parameter $y \in [0,1]^6$ are shown in Figure 19



(a) Sample 1

(b) Sample 2

Figure 19: Numerical solutions (Mach number) for the flow past an RAE airfoil, with two different realizations of the parameter y.

5.2.2 Generation of training data.

In order to generate the training, validation and test sets, we denote $\mathcal{Q}^{sob}$ as the first 1001 Sobol points on the domain $[0,1]^6$. For each point $y_j \in \mathcal{Q}^{sob}$, the maps or (rather their numerical surrogate) $\mathcal{L}^{\Delta}_{1,2}(y_j)$ is then generated from high-resolution numerical approximations $\mathbf{U}^{\Delta}(y)$ and the corresponding lift and drag are calculated by a numerical quadrature . For this numerical experiment, we take the first 128 Sobol points to generate the training set $\mathcal{S}$ and the next 128 points to generate the validation set $\mathcal{V}$. $\mathcal{Q}^{sob}$ forms the full test set $\mathcal{T}$.

5.2.3 Results of the Ensemble training procedure.

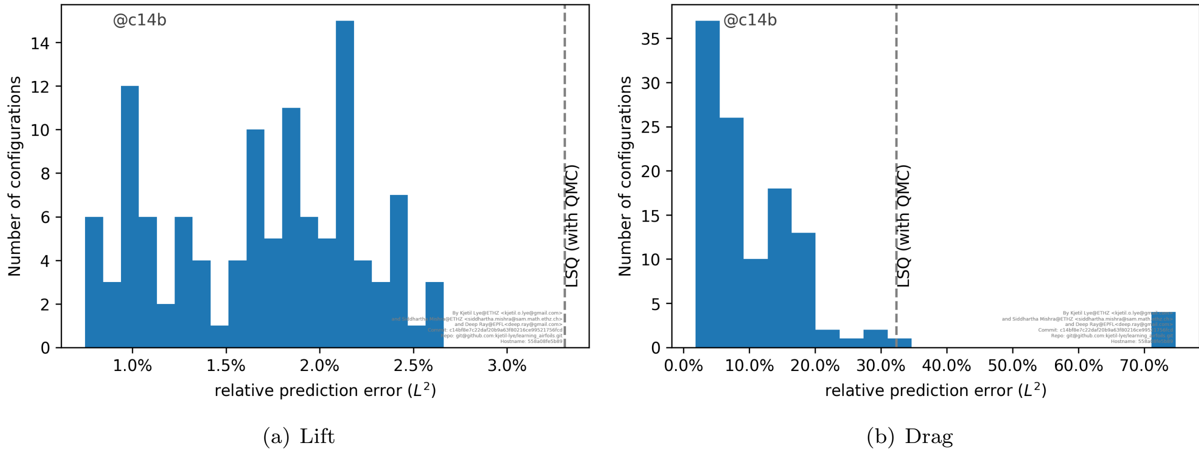
To set up the ensemble training procedure of section 4.2, we select a fully connected network, with number of layers and size of each layer listed in table 11. Our network has 1149 tunable parameters and this choice is clearly consistent with the prescriptions of Lemma 2.5. On this reference network architecture, we vary all the hyperparameters as described in section 4.2. This leads to 114 configurations (samples) over the hyperparameter space. Each sample is retrained 5 times by starting with different initial starting values for the optimization algorithm. Thus, we train a total of 570 networks for each observable, corresponding to different hyperparameter configurations, in parallel.

For each configuration, we select the retraining that minimizes the set selection criteria for the configuration. Once the network with best retraining is selected, it is evaluated on the whole test set $\mathcal{T} = \mathcal{Q}^{sob}$ and the percentage relative L^2 prediction error (2.27) computed. Histograms, describing the probability distribution over all hyperparameter samples for the lift and the drag are shown in figure 20. From this ensemble, we choose the hyperparameter configuration that led to the smallest prediction error and list them in table 12. As seen from this table, there is a difference in the best performing network corresponding to lift and to drag. However, both networks use ADAM as the optimizer and regularize with the L^2-norm of the weights in the loss function (2.12), even if the leading component of the loss function is MAE. Moreover, there are slight differences with the best performing networks for the Sod Shock tube problem (table 4).

We plot the training (and validation) loss, with respect to number of iterations (epochs) of ADAM, for the best performing networks and for the *effective network* (see table 5 for the underlying hyperparameters) in figure 21. As observed from the figure, the loss functions reduce by almost four orders of

Layer	Width (Number of Neurons)	Number of parameters
Hidden Layer 1	12	84
Hidden Layer 2	12	156
Hidden Layer 3	10	130
Hidden Layer 4	12	132
Hidden Layer 5	10	130
Hidden Layer 6	12	156
Hidden Layer 7	10	130
Hidden Layer 8	10	110
Hidden Layer 9	12	132
Output Layer	1	13
		1149

Table 11: Reference neural network architecture for the flow past airfoil problem.



(a) Lift

(b) Drag

Figure 20: Histograms depicting distribution of the percentage relative mean L^2 prediction error (2.27) (X-axis) over number of hyperparameter configurations (samples, Y-axis) for the lift and the drag in the flow past airfoil problem. Note that the range of X-axis is different in the plots.

Obs	Opt	Loss	L^1-reg	L^2-reg	Selection	*BP*. Err mean (std)	Ref. Err mean (std)
Lift	ADAM	MSE	0.0	7.8×10^{-6}	wass-train	0.786 (0.010)	0.931 (0.018)
Drag	ADAM	MAE	0.0	7.8×10^{-6}	mean-train	1.847 (0.022)	2.233 (0.010)

Table 12: The hyperparameter configurations that correspond to the best performing network (one with least mean prediction error) for the two observables of the flow past airfoils. The mean and standard deviation of the relative percentage L^2-prediction error for the best performing networks are compared with the corresponding errors (denoted by Ref. Err) of the *Effective network*, with hyperparameters listed in table 5

	Time (in secs)
Sample generation	24000
Training (Lift)	700
Evaluation ($\mathcal{L}_1$)	9×10^{-6}
Training (Drag)	840
Evaluation ($\mathcal{L}_2$)	10^{-5}

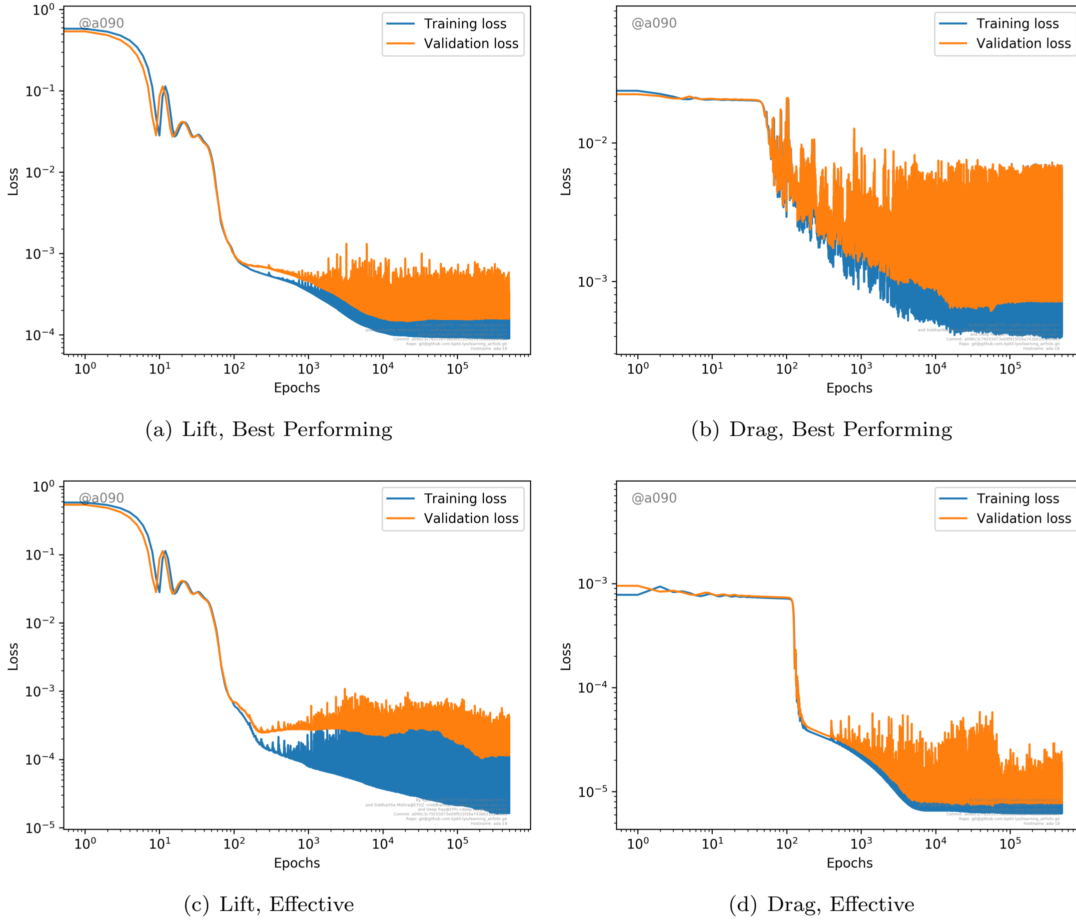
Table 13: Computational times (cost) of (single) sample generation, network training and (single) network evaluation, for the flow past airfoils problem. Each sample was generated using a parallelized finite-volume solver on 16 Intel(R) Xeon(R) Gold 5118 @ 2.30GHz processor cores. The entire run took 1500 secs on the cluster and translates into a total wall clock time of $1500 \times 16 = 24000$ secs. The training and evaluation of the neural networks were all performed on a Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz machine. The training and evaluation times are approximations of the average runtimes over all hyperparameter configurations.

magnitude during training for the lift whereas the reduction for the drag is about two orders of magnitude.

The prediction errors with the best performing and effective networks, are depicted in figure 22, in the form of a scatter plot for $\mathcal{L}_{1,2}(y)$ (X-axis) vs $\mathcal{L}^*_{1,2}(y)$ (Y-axis) for all $y \in \mathcal{T}$. As shown in this figure, the errors for lift and drag (with both the best performing and effective networks) are very low. This is further verified from table 12, from which we observe that the mean prediction errors with best performing (and effective) networks are less than *one percent* for the lift and approximately *two percent* for the drag. The attainment of such low predictions errors, even in this realistic problem, with a training set of merely $N = 128$ samples, is both unexpected and impressive, particularly when compared to the sum of sines experiment (see table 1). Theoretical considerations in section 2.3.3 estimated the generalization error by $\sqrt{\frac{M}{N}}$, with M, N being the number of parameters in the network and number of training samples respectively. Setting these numbers for this problem yields a relative error of approximately 300%. However, our prediction errors are $100 - 200$ times (two orders of magnitude) less, demonstrating the strength of trained neural network to generalize to unseen data. These results are further contextualized, when considered with reference to the computational costs, shown in table 13. In this table, we present the costs of generating a single sample, the average training time (over a subset of hyperparameter configurations) and the time to evaluate a trained network. Each sample is generated on the EULER high-performance cluster of ETH-Zurich. In this case, the training time is a very small fraction of the time to generate a single sample, let alone the whole training set whereas the computational cost of evaluating the neural networks is almost negligible Summarizing,*we demonstrate that deep neural networks can approximate observables of interest in this realistic problem, to a high accuracy, at a cost which is nine orders of magnitude less than a high-resolution CFD solver.*

The power of deep neural networks in this case is further highlighted when one compares them to a competing algorithm. One possible competitor is the classical *linear least squares algorithm* i.e, $\mathcal{L}^{\mathrm{lsq}}_j : \mathbb{R}^6 \to \mathbb{R}$ be affine map defined by

$$\mathcal{L}^{*,\mathrm{lsq}}_j = \arg\min \left\{ \sum_{y \in \mathcal{S}} (f(y) - \mathcal{L}_j(y))^2 \mid f : \mathbb{R}^6 \to \mathbb{R} \text{ is affine} \right\}, \quad j = 1, 2. \tag{5.8}$$



(a) Lift, Best Performing

(b) Drag, Best Performing

(c) Lift, Effective

(d) Drag, Effective

Figure 21: Training loss (2.31) and Validation loss (2.32) for the best performing (table 12) and effective (table 5) neural networks for the flow past airfoils problem as a function of the number of epochs.

The resulting convex optimization problem is solved by the standard gradient descent. The errors with the linear least squares algorithm are shown in figure 20. From this figure we can see that the prediction error with the linear least squares is approximately 4% for lift and 36% for drag. Hence, the best performing neural networks are approximately 5 times more efficient for lift and 20 times more efficient for drag, than the linear least squares algorithm. The gain over linear least squares implicitly measures how *nonlinear* the underlying map is. Clearly in this example, the drag is more nonlinear than the lift and thus more difficult to approximate. Nevertheless, the trained neural network approximates it to high accuracy.

5.2.4 Network sensitivity to hyperparameters.

We summarize the results of the sensitivity of network performance to hyperparameters below,

- *Overall sensitivity.* The overall sensitivity to hyperparameters is depicted as histograms, representing the distribution, over samples (hyperparameter configurations), shown in figure 20. As seen from the figure, the prediction errors with different hyperparameters are spread over a significant range for each observable. There are a few outliers which perform very poorly, particularly for the drag. On the other hand, quite a few hyperparameter configurations (samples) concentrate around the best performing network. Moreover all (most) of the hyperparameter choices led to better prediction errors for the lift (drag), than the linear least squares algorithm.
- *Choice of optimizer.* The difference between ADAM and the standard SGD algorithm is shown in figure 23. As in the Sod shock tube problem, ADAM is clearly superior to the standard SGD

Obs (Network)	Err (min)	Err (max)	Err (mean)	Err (std)
Lift (Best Performing)	0.786	1.336	1.004	0.187
Lift (Effective)	0.791	1.351	0.995	0.189
Drag (Best Performing)	1.847	8.016	3.841	2.332
Drag (Effective)	1.708	6.045	3.591	1.661

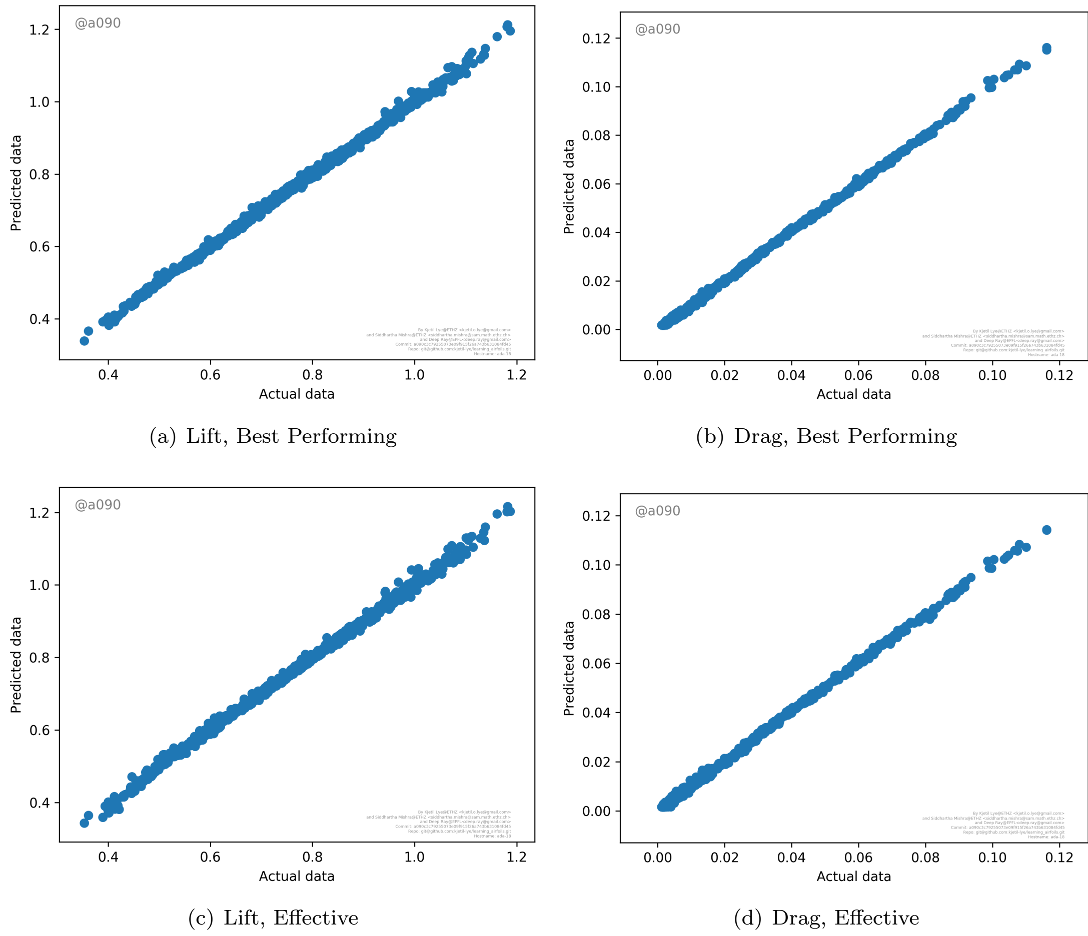
Table 14: Sensitivity of the best performing networks (listed in table 4) and the effective network (listed in table 5) to retrainings i.e starting values for ADAM, for the three observables in the Sod Shock tube. All errors are relative mean L^2 prediction error (2.27) in percentage and we list the minimum, maximum, mean and standard deviation of the error over 5 retrainings.

algorithm. Moreover, there are a few outliers with the ADAM algorithm that perform very poorly for the drag. As in the Sod shock tube problem and shown in figure 24 , these outliers correspond to the *bad set of ADAM*, defined in section 5.1.4.

- *Choice of loss function.* The difference between L^1 mean absolute error (MAE) and L^2 root mean square error (MSE) i.e, exponent $p = 1$ or $p = 2$ in the loss function (2.11) is plotted in the form of a histogram over the error distributions in figure 25. The difference in performance, with respect to choice of loss function is minor. However, there is an outlier, based on MSE, for the drag.
- *Choice of type of regularization.* The difference between choosing an L^1 or L^2 regularization term in (2.12) is shown in figure 26. Again, the differences are minor but the L^2 regularization seems to be slightly better, particularly for the drag.
- *Value of regularization parameter.* The variation of network performance with respect to the value of the regularization parameter λ is shown in figure 27. We observe from this figure that a little amount of regularization works better than no regularization. However, the variation in prediction errors wrt respect to the non-zero values of λ is minor.
- *Choice of selection criteria.* The distribution of prediction errors with respect to the choice of selection criteria for retraining, namely *train,val,mean-train,wass-train* is shown in figure 28. As in the case of the Sod shock tube problem, there are very minor differences between *mean-train* and *wass-train.* On the other hand, *wass-train* is clearly superior to the other two selection criteria.
- *Sensitivity to retrainings.* The variation of performance with respect to retrainings i.e, different random initial starting values for ADAM, is provided in table 14. In this table, we list the minimum, maximum, mean and standard deviation of the relative prediction error, over 5 retrainings, for the best performing networks and the effective network for each observable. As seen from this table, the sensitivity to retraining is rather low for the lift. On the other hand, it is more significant for the drag, particularly for the best performing network. However, for the effective network for the drag, the standard deviation (in retrainings) of the error is still about 40% of the mean.
- *Variation of Network size.* We consider the dependence of performance with respect to variation of network size by focussing on the effective network (table 5) and varying the width and depth for this network configuration. The resulting prediction errors are shown in table 15 where a 3×3 matrix for the errors (rows represent width and columns depth) is tabulated. All the network sizes are consistent with the requirements of lemma 2.5. As observed from the table, the sensitivity to network size is greater for the drag than for the lift or for the Sod shock tube problem (table 8). It appears that increasing the width for constant depth results in lower errors for the drag. On the other hand and as in the case for the Sod Shock tube problem, training clearly failed for some of the larger networks.

5.2.5 Network performance with respect to number of training samples.

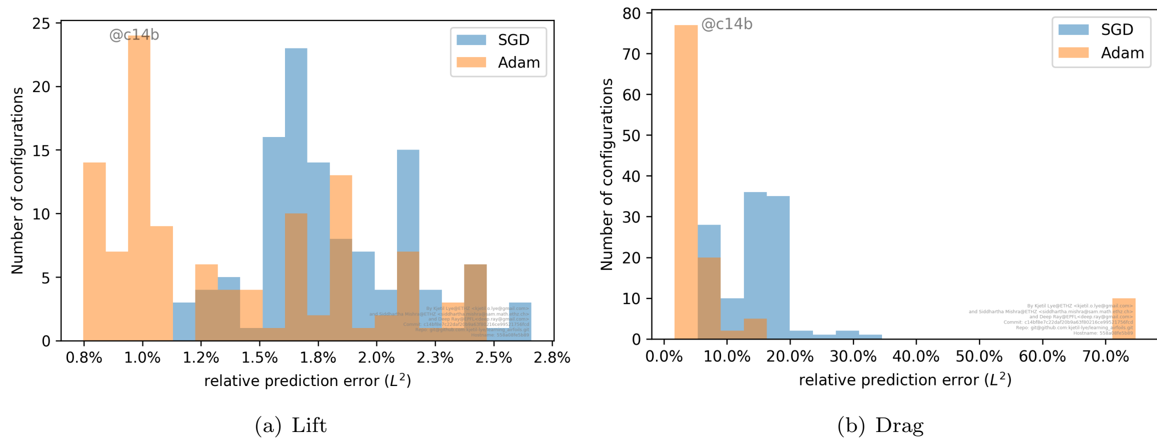
All the above results were obtained with $N = 128$ training samples. We have repeated the entire ensemble training with different samples sizes i.e $N = 32, 64, 256$ also and present a summary of the results in figure



(a) Lift, Best Performing (b) Drag, Best Performing

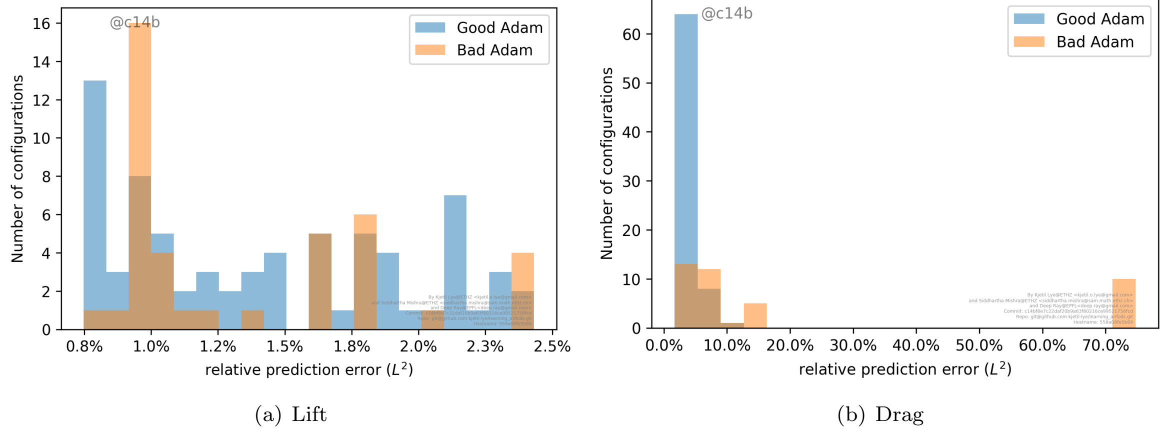
(c) Lift, Effective (d) Drag, Effective

Figure 22: Scatter plots depicting prediction errors for the lift and the drag of the flow past airfoils problem. X-axis (Test data), Y-axis: Predictions by best performing or effective neural networks.



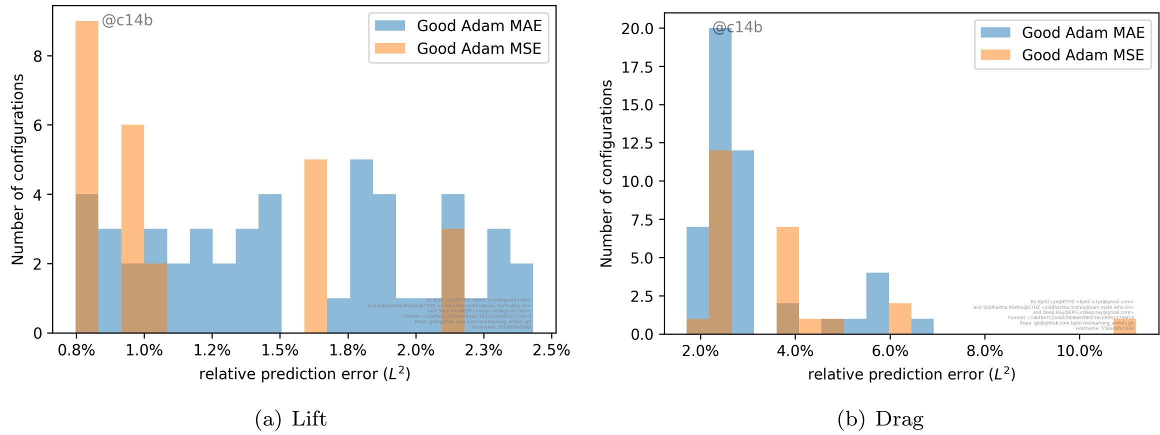
(a) Lift (b) Drag

Figure 23: Histograms for the relative prediction errors comparing the ADAM and SGD algorithms on the flow past airfoils.



(a) Lift (b) Drag

Figure 24: Histograms of the relative prediction errors, comparing *Good* ADAM and *bad* ADAM, for the flow past airfoils.

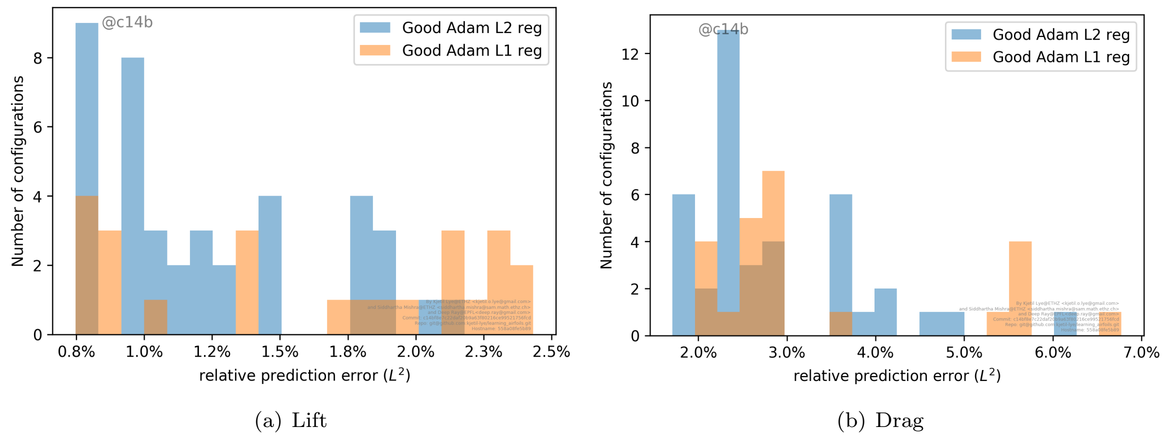


(a) Lift (b) Drag

Figure 25: Histograms for the relative prediction errors, comparing the mean absolute error and mean square error loss functions, for the flow past airfoils

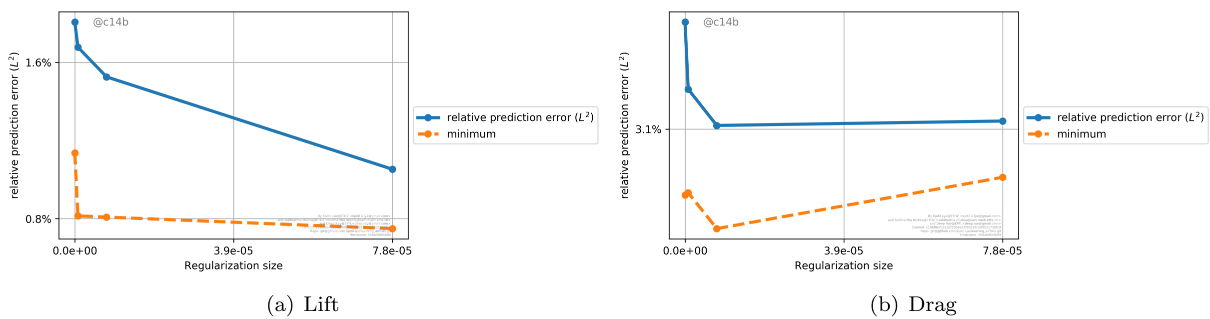
Width/Depth	6	12	24
4	$[1.21, 4.37]$	$[0.83, 1.74]$	$[1.09, 2.48]$
8	$[0.84, 3.66]$	$[0.86, 2.02]$	$[0.81, 1.69]$
16	$[0.80, \neg]$	$[\neg, 74.72]$	$[\neg, 5.11]$

Table 15: Performance of trained neural networks with respect to the variation of network size for the flow past airfoils problem. The rows represent variation with respect to Width (number of neurons per hidden layer) and the columns represent variation with respect to Depth (number of hidden layers). For each entry of the Width-Depth matrix, we tabulate the vector of relative percentage mean prediction error for the effective network (see table 5). The components of each vector represent error in [Lift, Drag] and $\neg$ is used to indicate that the training procedure failed for this particular configuration.



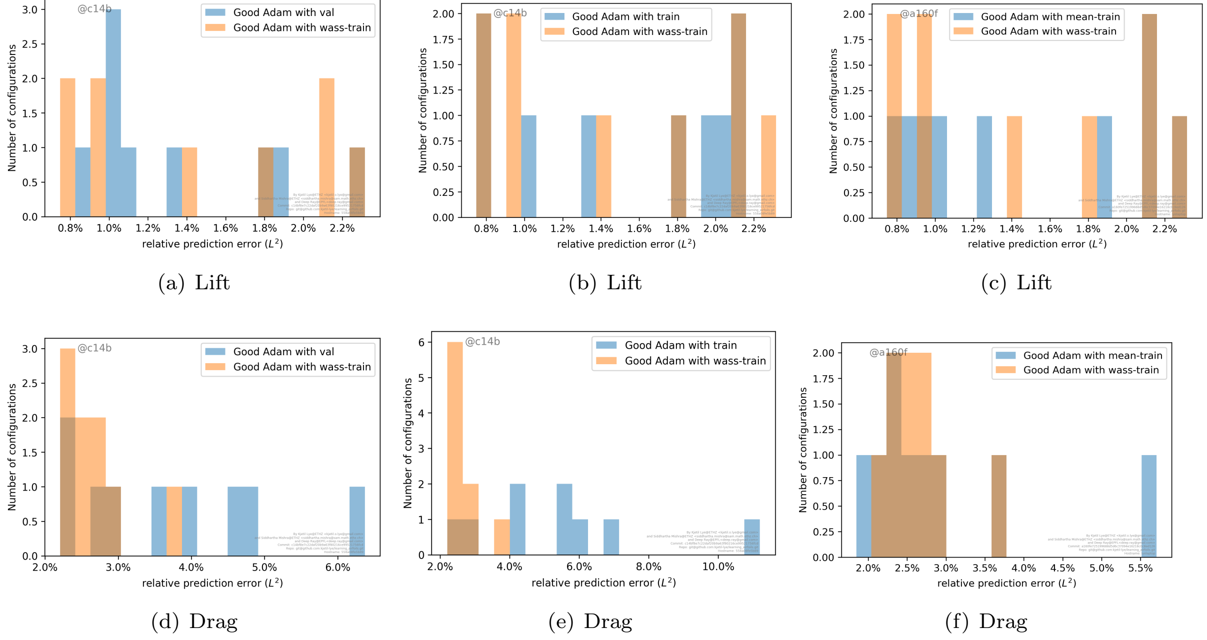
(a) Lift (b) Drag

Figure 26: Histograms for the relative prediction error, comparing L^1 and L^2 regularizations of the loss function, for the flow past airfoils



(a) Lift (b) Drag

Figure 27: Variation of prediction error (Y-axis) with respect to the size of the regularization parameter λ in (2.12) (X-axis) for the flow past airfoils problems. The minimum and mean of prediction error (over all hyperparameter configurations) is shown



(a) Lift (b) Lift (c) Lift

(d) Drag (e) Drag (f) Drag

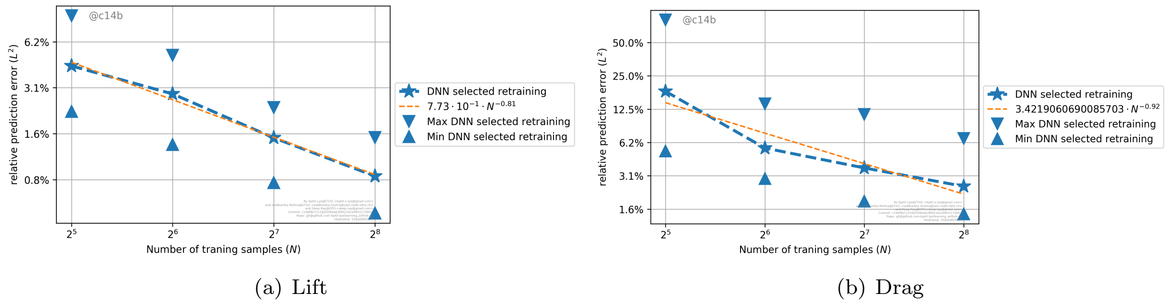
Figure 28: Network performance with respect to selection criteria for retrainings (see section 4.2 i.e, *train,val,mean-train,wass-train* for the flow past airfoils problem. We compare *wass-train* with each of the other three criteria. Histograms for prediction error (X-axis) with number of configurations (Y-axis) are shown.

29. In this figure, we plot the mean (percentage relative) L^2 prediction error with respect to the number of samples. For each sample size, three different errors are predicted, namely the minimum (maximum) prediction errors and the mean prediction over a subset of the hyperparameter space. This subset consists of the *good set of ADAM* as the optimizer , while varying the type of loss function, type of regularization, the value of regularization parameter and the selection criteria. Moreover, we only consider errors with selected (optimal) retrainings in each case. As seen from the figure, the prediction error *decreases with sample size N* as predicted by the theory. The decay seems of the of form $\left(\frac{U_i}{N}\right)^{\alpha_i}$ with $\alpha_1 = 0.81$ and $U_1 = 0.77$ for the lift and $\alpha_2 = 0.92$ and $U_2 = 3.42$ for the drag. In both cases, the decay of error with respect to samples, is at a higher rate than that predicted by the theory in (2.28). Moreover, the constants are much lower than the number of parameters in the network. Both facts indicate a very high degree of compressibility (at least in this possible pre-asymptotic regime) for the trained networks and explains the unexpectedly low prediction errors.

5.2.6 UQ with deep learning

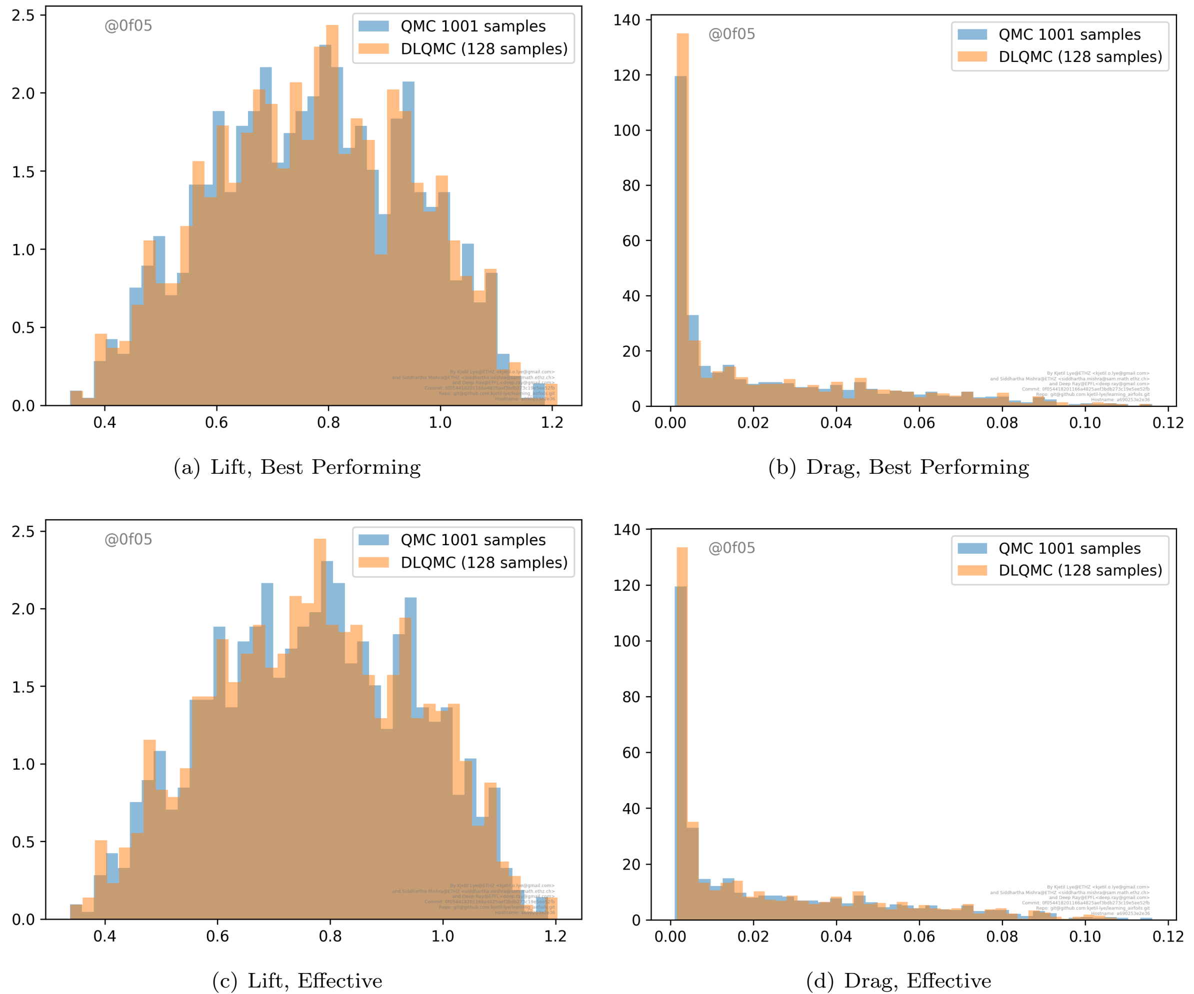
In this section, we will approximate the underlying probability distributions (measures) (3.33), with respect to the lift and the drag in the flow past airfoils problem. A reference measure is computed from the whole test set of 1001 samples and the corresponding histograms are shown in figure 30 to visualize the probability distributions. We approximate these distributions with the DLQMC algorithm 3.5. The corresponding histograms, computed with the best performing networks for each observable (listed in table 12) and the effective network (see table 5) are also shown in figure 30. The figures show that the histograms computed with the neural networks match the reference histograms to very high accuracy. Hence for both lift and drag, there is a very good qualitative agreement between the outputs of the DLQMC algorithm and the reference measure.

In order to quantify the gain in computational efficiency with the DLQMC algorithm over the baseline QMC algorithm in approximating probability distributions, we follow the procedure outlined for the Sod shock tube problem in section 5.1.6. To this end, we see from figure 31 that the baseline QMC algorithm



(a) Lift (b) Drag

Figure 29: Relative percentage mean square prediction error (2.27) (Y-axis) for the neural networks approximating the flow past airfoils problem, with respect to number of training samples (X-axis). For each number of training samples, we plot the mean, minimum and maximum (over hyperparameter configurations) of the prediction error. Only the selected (optimal) retraining is shown.



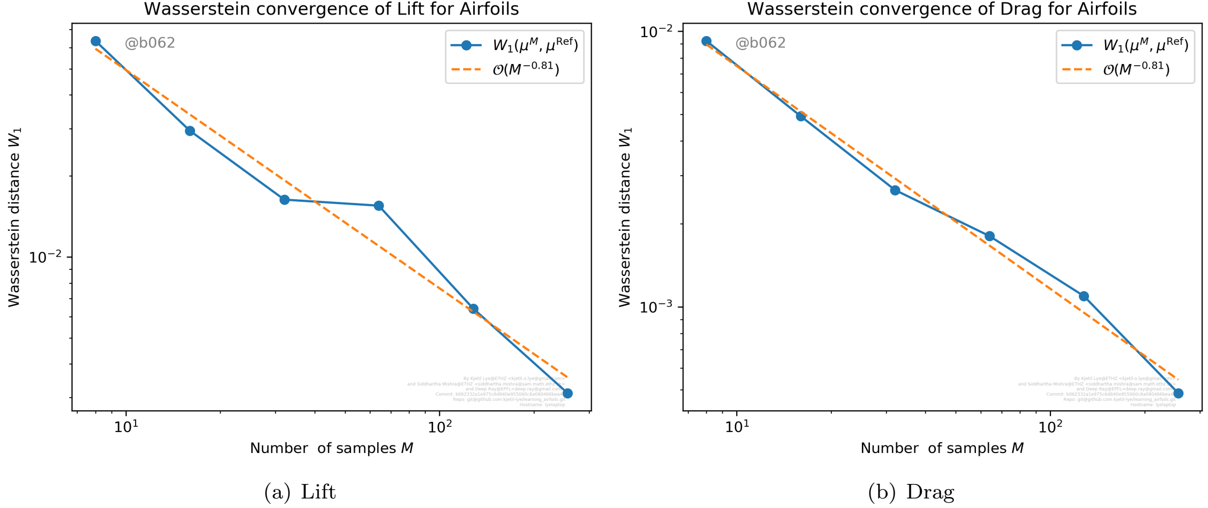
(a) Lift, Best Performing (b) Drag, Best Performing

(c) Lift, Effective (d) Drag, Effective

Figure 30: Empirical histograms representing the probability distribution (measure) for the lift and the drag in the flow past airfoils problem. We compare the reference histograms (computed with the test set $\mathcal{T}$) and the histograms computed with the DLQMC algorithm.

Observable	Network	Speedup
Lift	Best Performing	6.64
Lift	Effective	5.21
Drag	Best Performing	8.56
Drag	Effective	6.91

Table 16: Real speedups (5.5), for the lift and the drag in the flow past airfoils problem, comparing DLQMC with baseline QMC algorithm. Speedups for both best performing (table 12) and effective (table 5) networks are shown.



(a) Lift

(b) Drag

Figure 31: Convergence of the Wasserstein distance $W_1\left(\hat{\mu}_{i,qmc}, \hat{\mu}_i^{\Delta}\right)$ (Y-axis) for the of the lift and the drag for the flow past airfoils, with respect to number of QMC points (X-axis)

has a rate of convergence of 0.81 for both lift and drag. We use these in formulas (5.4) and (5.5) to compute the *real speed up* that we obtain by using the DLQMC algorithm, over the baseline QMC algorithm, in approximating probability distributions. The speedups are shown in table 16. We observe from this table that we obtain speedups ranging between half an order to an order of magnitude for the lift and the drag. The speedups for drag are slightly better than that for lift. We would like to point out that in this case, the assumption of ignoring the computational cost of training while estimating the error is completely justified. As seen from table 13, the training time is approximately 30 times less than the cost of generating a single sample. Hence, the training costs are negligible when compared to the total cost of generating 128 samples.

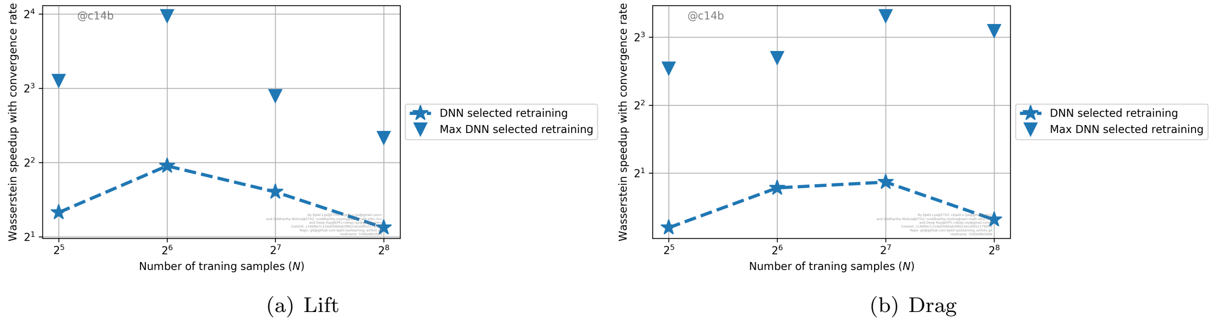
In figure 32, we plot the speedup of the DLQMC algorithm (over the baseline QMC algorithm) as a function of the number of training samples in the deep learning algorithm 2.1. As seen from the figure, the best speed ups are obtained in case of 64 training samples for the lift and 128 training samples for the drag. This is on account of a complex interaction between the prediction error which decreases (rather fast) with the number of samples and the fact that the errors with the baseline QMC algorithm also decay with an increase in the number of samples.

5.2.7 Comparison with Monte Carlo algorithms

We have computed a Monte Carlo approximation of the probability distributions of the lift and the drag by randomly selecting $N = 128$ points from the parameter domain $Y = [0,1]^6$ and computing the probability measure $\hat{\mu}_{mc} = \frac{1}{N}\sum_{j=1}^{N}\delta_{\mathcal{L}^{\Delta}(y_j)}$. We compute the Wasserstein error $W_1(\hat{\mu}_{mc}, \hat{\mu}^{\Delta})$, with respect to the reference measure $\hat{\mu}^{\Delta}$ and divide it with the error obtained with the DLQMC algorithm to obtain a *raw speedup* of the DLQMC algorithm over MC. As MC errors converge as a square root of the number

Observable	Network	Speedup over MC
Lift	Best Performing	246.02
Lift	Effective	166.09
Drag	Best Performing	179.54
Drag	Effective	126.91

Table 17: Real speedups, for the lift and the drag in the flow past airfoils problem, comparing DLQMC with the baseline MC algorithm. Speedups for both best performing (table 12) and effective (table 5) networks are shown.



(a) Lift (b) Drag

Figure 32: Real speedups (5.5) for the DLQMC algorithm over the baseline QMC algorithm (Y-axis) with respect to number of training samples (X-axis) for the flows past airfoils problem. The maximum and mean speed up (over all hyperparameter configurations) are shown.

of samples, the *real speedup* of DLQMC over MC is calculated by squaring the raw speedup. We present this real speedup over MC, with respect to both best performing and effective networks in table 17. We observe from this table that the speedups of the DLQMC algorithm over the baseline MC algorithm are very high and amount to at least two orders of magnitude. This is not surprising as the baseline QMC algorithm significantly outperforms the MC algorithm for this problem, see forthcoming paper [30] for further comparisons. Given that the DLQMC algorithm was an order of magnitude faster than the baseline QMC algorithm, the cumulative effect leads to a two orders of magnitude gain over the baseline MC algorithm.

5.2.8 Choice of training set.

As mentioned in sections 2 and 3, the training set $\mathcal{S}$ can be chosen to be either randomly chosen (Monte Carlo) points in the parameter space Y or they can be low-discrepancy sequences (Quasi-Monte Carlo points). All the results presented thus far are based on choosing QMC (Sobol) points as the training set. For the flow past airfoils problem, we chose $N = 128$ random points in Y as the training set. With this training set, we repeated the ensemble training procedure for deep neural networks to approximate the lift and the drag and found the best performing networks. The relative percentage mean L^2-prediction error (2.27) (with respect to a test set of 320 random points) was computed and is presented in table 18. As seen from this table, the prediction errors with respect to the best performing networks are considerably (an order of magnitude) higher than the corresponding errors obtained for the QMC training points (compare with table 12). Hence, at least for this problem, Sobol points are a much better choice for the training set than random points. An intuitive reason for this could lie in the fact that the Sobol points are better distributed in the parameter domain than random points. So on an average, a network trained on them will generalize better to unseen data, than it would if trained on random points.

Obs	Opt	Loss	L^1-reg	L^2-reg	Selection	*BP*. Err mean	Ref. Err mean
Lift	SGD	MAE	0.0	7.8×10^{-7}	mean-train	8.487	10.40
Drag	ADAM	MAE	7.8×10^{-7}	0.0	train	20.25	29.36

Table 18: The hyperparameter configurations that correspond the best performing network (one with least mean prediction error) for the two observables of the flow past airfoils, trained on random (Monte Carlo) training set. The mean of the relative percentage L^2-prediction error for the best performing networks are compared with the corresponding mean error (denoted by Ref. Err) of the *Effective network*, with hyperparameters listed in table 5.

6 Discussion

The goal in many CFD simulations is the computation of observables (functionals or quantities of interest) for the solutions of nonlinear PDEs that govern fluid flows. Many interesting problems in CFD such as uncertainty quantification (UQ), Bayesian inversion, optimal control and shape optimization require a large number of forward solves of the underlying PDE in order to compute (statistics of) the observable. However, each high-resolution CFD solve can be expensive, even on state of the art HPC systems. Hence, the afore mentioned problems are considered to be prohibitively computationally expensive.

Instead of computing every realization of the underlying input parameters to observable map (2.4), we propose in this paper, to harness the power of machine learning and *learn* or approximate this map by a deep artificial neural network of the form (2.6). However and as explained in section 2.3, the task of finding the correct network architecture and training a network that generalizes well is very challenging in this context, on account of the following hurdles,

- The underlying parameters to observable map (2.3) is not very regular (smooth) for most problems of interest in fluid dynamics. A priori, one expects that the underlying map is at most Lipschitz continuous (see theorem 2.4). Consequently, applying results on approximation of functions by neural networks (see theorem 2.2) leads to unrealistically large sizes for the approximating networks. Such networks are hard to train and very expensive to compute.

- The generalization error (2.28) scales as a fractional negative power of the number of training samples. Even with reasonable upper bounds in (2.28), one would need a large number, around $O(10^4)$, of training samples in order to obtain errors of acceptable magnitude i.e, approximately one percent relative error. Computing such large number of training samples is very expensive and hence, unrealistic in practice.

Thus, we are confronted with the challenge of *finding deep neural networks that can accurately approximate maps of low regularity in a relatively data poor regime*. This is in stark contrast to many common *big data* applications of deep learning. An illustration of some of these difficulties is provided with the sum of sines example (see table 1 and figure 2) where trained neural networks approximated a high-dimensional sinusoidal function with unacceptably large errors.

We attempt to overcome this challenge by a combination of the following novel ideas,

- We focus on learning observables of the form $\mathcal{L}$ (2.3), rather than learning fields, $\mathbf{U}(t, x, y)$ that solve the underlying parameterized convection-diffusion PDEs (2.1). A key reason for this choice is the fact that map $y \mapsto \mathcal{L}(y)$ can be more regular than the mapping $(t, x, y) \mapsto \mathbf{U}(t, x, y)$. This is already indicated by recognizing that the observable $\mathcal{L}$ in Theorem 2.4, for a scalar conservation law is Lipschitz continuous whereas the field $\mathbf{U}$ is only BV. One can expect that observables, defined by averaging, may possess better compression properties than fields. This focus on observables distinguishes our work from many recent papers on the application of deep learning to PDEs, see [37, 38, 11, 12] and references therein, where the objective was to approximate solution fields.

- We propose that the training data in the deep learning algorithm 2.1 be chosen as low-discrepancy sequences, such as Sobol or Halton sequences, that are heavily used in Quasi-Monte Carlo methods. Our choice can be contrasted to the standard practice in machine learning where random points are selected for generating training data [44]. We expect that the equi-distribution properties of

low-discrepancy sequences will lead to the networks, trained on them, to generalize better than those trained on random points. This intuition is indeed justified in numerical experiments (see table 18), in which we obtain an order of magnitude smaller errors with networks trained on QMC points than on MC points. Moreover, such low-discrepancy sequences are very easy to generate and do not suffer from the curse of dimensionality, at least for moderate to high dimensional parameter spaces.

- There are quite a few hyperparameters in training deep neural networks and some of them are listed in table 2. Given the challenge of finding networks that generalize well in the data poor regime that we are working in, we have proposed an *ensemble training procedure* in section 4.2, wherein a whole ensemble of hyperparameter configurations are trained in parallel. We use this procedure for finding the *best performing* networks as well as for studying the sensitivity of network performance to hyperparameter choice. This systematic screening of hyperparameters is essential in our context as theory provides very little guidance on which network architectures and hyperparameter configurations, to use.

- We demonstrate the utility of our deep learning algorithm 2.1 in a very challenging UQ setting, namely that of computing probability distributions (3.33) of the observables. The efficient computation of probability distributions is considered to be significantly harder than computing statistical moments. Under the assumption that the underlying network generalizes well, we prove in Theorem 3.11 that the resulting DLQMC algorithm 3.5 will be more efficient (provide a speed up) with respect to the baseline Quasi-Monte Carlo (QMC) algorithm.

The proposed algorithms and theoretical predictions are tested on two representative numerical experiments. Both of them consider the compressible Euler equations (4.1). One of them, the stochastic Sod shock tube, is a one-dimensional academic problem whereas the second one deals with flows past a RAE2822 airfoil and is a benchmark realistic problem [23]. From the numerical experiments, we find that,

- Our deep learning algorithm resulted in very low prediction errors, of less than 2% relative error for all the observables that we tested. This is in stark contrast to the baseline established by the sum of sines example and is far better than what the theory predicts. In fact, a posteriori, we could calculate that the trained networks have a very high degree of compressibility (in the sense of [2]) and resulted in prediction errors that are two to three orders of magnitude better than expected. Moreover, we were able to approximate lift and drag, to high accuracy, but at a cost that is *nine orders of magnitude* less than a high-resolution CFD simulation.

- Our ensemble training algorithm revealed that network performance was only sensitive to a few hyperparameters, In particular, ADAM was clearly preferred as an optimizer over standard SGD whereas the exact form of the loss function and regularization terms did not matter much. Similarly, it was better to add a small amount of regularization in the loss function (2.12) whereas the exact amplitude of this term was not very pertinent to network performance. A crucial aspect was the use of *retrainings* i.e, multiple (random) starting points for the ADAM (SGD) algorithm. Network performance depended on these retrainings and a good criteria for selecting the best retraining from the data at hand, was important. We proposed statistical criteria such as *wass-train* or *mean-train* that led (on an average) to better generalization. Similarly, network size did not seem to matter much as long as the trained network had enough width and depth to be consistent with the number of parameters prescribed in lemma 2.5.

  Based on the results of the ensemble training procedure, we suggest the following hyperparameter configuration i.e *any network with ADAM optimizer, either L^1 or L^2 loss functions and regularization terms and a very small amount of regularization and with a network size (width and depth) of* $O(d + N)$ generalized well, provided that there are *enough retrainings and the optimal network is selected according to mean-train or wass-train.* All these criteria are met by the so-called *effective network*, listed in table 5, and this could be a network architecture of choice in many problems.

- The DLQMC algorithm 3.5 that combined deep learning and Quasi-Monte Carlo was found to be very efficient at uncertainty propagation, particularly for the intricate problem of computing

probability distributions. It provided around an order of magnitude speedup over the baseline QMC algorithm and atleast two orders of magnitude speedup over the baseline Monte Carlo (MC) algorithm.

- We also observed that specifying the number of training samples (size of training set $\mathcal{S}$) can be tricky. The prediction error decreases with the number of training samples (check from figures 13 and 29) On the other hand, the error of baseline QMC method also reduces by increasing the number of samples (figures 15 and 31). Thus, balancing these two factors is critical in determining the optimal number of training samples in the context of UQ. From figure 32, we see that $N = 64$ or $N = 128$ are good choices at least for the flows past airfoil problem. This nicely coincides with the practical consideration of only being able to generate about $O(100)$ training samples within a realistic computational cost.

Summarizing, we propose a recipe for choosing neural network architectures and hyperparameters that can provide very low prediction errors for approximating maps of low regularity, while being trained on a relatively few training samples. The training data should be sampled from consecutive QMC points and network hyperparameters correspond to those listed in table 5. The network size (depth and width) should comply with the requirements of lemma 2.5. Once the network architecture and hyperparameter configuration is set, the network should be retrained a few times and the optimal retraining chosen, based on some statistical criteria such as *wass-train*. Based on the evidence gleaned from our numerical experiments, we expect that this recipe will be very efficient in the context to finding neural networks for problems that involve approximating observables of solutions to nonlinear PDEs.

The main limitation of the current paper is the lack of rigorous theoretical guarantees on the generalization error (2.26). Obtaining such bounds are considered to be a major open question in theoretical machine learning, [2] and references therein. On the other hand, we present numerical evidence to suggest that there is a very high degree of compressibility in the problems that we consider. Proving such bounds is a subject of current investigation.

The algorithms of this paper can be extended in the following directions,

- Our training data is generated from high-resolution CFD simulations. However, the approach readily fits with the availability of observational (measurement) data as a substitute or as augmentation of data generated by simulations. If such data is available, then it can be added to the training set in the deep learning algorithm 2.1. Presumably, the data needs to be denoised first, for instance by another neural network. The addition of such data will only increase the accuracy of our algorithm.

- Our proposed approach can be thought of as an example of *model order reduction* (MOR) [49]. The neural network $\mathcal{L}^*$ is the reduced order model (ROM). Generation of training data and training the neural network is the *offline* step and evaluation of the neural network is the *online* step. We would like to emphasize that at least for hyperbolic PDEs, our neural network seems to significantly outperform available reduced order models (see [1] and references therein). The accuracy of the neural network is at least comparable, if not higher, whereas the cost when compared to standard ROMs is several orders of magnitude lower. More careful comparison between these two approaches is certainly warranted.

- The proposed approach can readily be extended to approximating solution fields $\mathbf{U}(t, x, y)$ of (2.1), but with the caveat that the field might be less regular (or less compressible) than observables.

- In the current paper, our numerical examples focused on the compressible Euler equations. However, no special feature of this particular PDE is used. In fact, the deep learning algorithm 2.1 and the DLQMC algorithm 3.5 are stated for any general PDE (2.1) with a very general form of the observable (2.3). As such, they should be readily extendable to other PDEs in CFD, such as the incompressible Navier-Stokes equations and also to turbulence models (where the observables may also be time and ensemble averages).

  The extension to other classes of PDEs such as elliptic and parabolic PDEs is also straightforward. In fact, given that the solutions of such PDEs are more regular, the task of finding appropriate neural networks should be easier, see [43] and references therein.

- In the current paper, we only applied the deep learning algorithm 2.1 in the context of forward UQ. The fact that the trained neural networks approximate the underlying observables to high accuracy, but at very low computational cost paves the way for using this algorithm in a variety of contexts. In particular, the neural network can serve as an *accurate surrogate* for MCMC algorithms within Bayesian inversion and data assimilation (filtering) . Moreover, it can be very useful in the context of surrogate based optimization methods. In particular, the gradient descent step in any optimization algorithm will be computed very efficiently, aided by the fact that the derivatives of the output of the trained neural network with respect to input parameters, can be computed by backpropagation. Applications of our algorithms to these problems is the subject of future work.

Acknowledgements

The research of SM is partially support by ERC Consolidator grant (CoG) NN 770880 COMANFLO. A large proportion of computations for this paper were performed on the ETH compute cluster EULER.

References

[1] R. Abgrall and R. Crisovan. Model reduction using L1-norm minimization as an application to nonlinear hyperbolic problems. *Internat. J. Numer. Methods Fluids* 87 (12), 2018, 628–651.

[2] 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.

[3] A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Trans. Inform. Theory,* 39(3), 930-945, 1993.

[4] H. Bijl, D. Lucor, S. Mishra and Ch. Schwab. (editors). *Uncertainty quantification in computational fluid dynamics.,* Lecture notes in computational science and engineering 92, Springer, 2014.

[5] H. Bölcskei, P. Grohs, G. Kutyniok, and P. Petersen. Optimal approximation with sparsely connected deep neural networks. *arXiv preprint*, available from arXiv:1705.01714, 2017.

[6] A. Borzi and V. Schulz. *Computational optimization of systems governed by partial differential equations.* SIAM (2012).

[7] H. J. Bungartz and M. Griebel. Sparse grids. *Acta Numer.*, 13, 2004, 147-269.

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

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

[10] Constantine M. Dafermos. *Hyperbolic Conservation Laws in Continuum Physics (2nd Ed.).* Springer Verlag (2005).

[11] 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.

[12] 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.

[13] 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.

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

[15] U. S. Fjordholm, R. Käppeli, S. Mishra and E. Tadmor, Construction of approximate entropy measure valued solutions for hyperbolic systems of conservation laws. *Found. Comput. Math.,* 17 (3), 2017, 763-827.

[16] U. S. Fjordholm, K. O. Lye, S. Mishra and F. R. Weber, Numerical approximation of statistical solutions of hyperbolic systems of conservation laws. *In preparation*, 2019.

[17] R. Ghanem, D. Higdon and H. Owhadi (eds). *Handbook of uncertainty quantification*, Springer, 2016.

[18] I. Goodfellow. Y. Bengio and A. Courville. *Deep learning,* MIT Press, (2016), available from `http://www.deeplearningbook.org`

[19] Edwige Godlewski and Pierre A. Raviart. *Hyperbolic Systems of Conservation Laws.* Mathematiques et Applications, Ellipses Publ., Paris (1991).

[20] I. G. Graham, F. Y. Kuo, J. A. Nichols, R. Scheichl, Ch. Schwab and I. H. Sloan. Quasi-Monte Carlo finite element methods for elliptic PDEs with lognormal random coefficients. *Numer. Math.* 131 (2), 2015, 329–368.

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

[22] J S. Hesthaven. *Numerical methods for conservation laws: From analysis to algorithms.* SIAM, 2018.

[23] 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.

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

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

[26] 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.

[27] L. D. Landau and E. M. Lipschitz. Fluid Mechanics, 2nd edition, *Butterworth Heinemann,* 1987, 532 pp.

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

[29] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations, steady state and time dependent problems.* SIAM (2007).

[30] K. O. Lye, S. Mishra and D. Ray. Multi-level Monte Carlo and Quasi-Monte Carlo methods for efficient uncertainty quantification in aerodynamics. *In preparation,* 2019.

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

[32] 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.

[33] H. N. Mhaskar. Neural networks for optimal approximation of smooth and analytic functions, *Neural Comput.*, 8 (1), 1996, 164-177.

[34] J. Munkres. Algorithms for the Assignment and Transportation Problems *Journal of the Society for Industrial and Applied Mathematics*, 5 (1), 1957, 32–38.

[35] 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.

[36] P. Petersen and F. Voightlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *ArXIV preprint*, available from arXiv:1709.05289, 2018.

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

[38] 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.

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

[40] 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.

[41] D. Ray. Entropy-stable finite difference and finite volume schemes for compressible flows, *Doctoral thesis*, 2017, available from `https://deepray.github.io/thesis.pdf`

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

[43] C. Schwab and J. Zech. Deep learning in high dimension. *Technical Report 2017-57*, Seminar for Applied Mathematics, ETH Zürich, 2017.

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

[45] A. M. Stuart. Inverse problems: a Bayesian perspective. *Acta Numerica,* 19, 2010, 451-559.

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

[47] L. N. Trefethen. *Spectral methods in MATLAB*, SIAM, (2000).

[48] F. Troltzsch. *Optimal control of partial differential equations.* AMS, (2010).

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

[50] C. Villani. *Topics in Optimal Transportation.* American Mathematical Society, Graduate Studies in Mathematics, Vol. 58 (2013)

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

[52] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *In International Conference on Learning Representations,* 2017.

[53] Linfeng Zhang, De-Ye Lin, Han Wang, Roberto Car, and Weinan E. Active Learning of Uniformly Accurate Inter-atomic Potentials for Materials Simulation. *ArXIV preprint*, available from arXiv:1810.11890, 2018.

[54] Chollet, François and others Keras `https://keras.io`, 2015.

[55] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlen, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu and Xiaoqiang Zheng TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems `https://www.tensorflow.org/`, 2015.