Enhancing accuracy of deep learning algorithms by training with low-discrepancy sequences

S. Mishra and TK. Rusch

___

Enhancing accuracy of deep learning algorithms by training with low-discrepancy sequences

Siddhartha Mishra [*] and T. Konstantin Rusch [†]

May 26, 2020

Abstract

We propose a deep supervised learning algorithm based on low-discrepancy sequences as the training set. By a combination of theoretical arguments and extensive numerical experiments we demonstrate that the proposed algorithm significantly outperforms standard deep learning algorithms that are based on randomly chosen training data, for problems in moderately high dimensions. The proposed algorithm provides an efficient method for building inexpensive surrogates for many underlying maps in the context of scientific computing.

1 Introduction

A fundamental objective of computer simulation is the *prediction* of the response of a physical or biological system to inputs. This is formalized in terms of *evaluation* of an underlying function (map, observable) for different inputs, i.e.

$$\text{Compute } \mathcal{L}(y), \quad \text{for } y \in Y.$$

Here, the underlying function $\mathcal{L} : Y \mapsto Z$, for some $Y \subset \mathbb{R}^d$, possibly for $d >> 1$ and Z is a finite or even an infinite dimensional Banach space.

For many interesting systems in physics and engineering, the evaluation of the underlying map $\mathcal{L}$ requires the (approximate) solution of ordinary or partial differential equations. Prototypical examples for such $\mathcal{L}$'s include the response of an electric circuit to a change in input current, the change in bulk stresses on a structure on account of the variation of the load, the mean global sea surface temperatures that result from different levels of CO_2 emissions and the lift and the drag across the wing of an aircraft for different operating conditions such as mach number and angle of attack of the incident flow.

The cost of computing the underlying map $\mathcal{L}$ for different inputs can be very high, for instance when evaluating $\mathcal{L}$ requires computing solutions of PDEs in several space dimensions. Moreover, a large number of problems of interest are of the *many query* type, i.e. many different instances of the underlying map have to be evaluated for different inputs. These problems can arise in the context of prediction (predict the wave height at a buoy when the tsunami is triggered by an earthquake with a certain initial wave displacement), uncertainty quantification (calculate statistics of $\mathcal{L}$ for uncertain inputs such as the mean and the variance of the stress due to random loading), optimal design and control (design the wing shape to minimize drag for constant lift) and (Bayesian) inverse problems (calibrate parameters in a climate model to match observed mean sea surface temperatures). The computational cost of such many query problems, requiring a large number of computational PDE solves, can be prohibitively expensive, even on state of the art high-performance computing (HPC) platforms.

One paradigm for reducing the computational cost of such many query problems consists of generating *training data*, i.e. computing $\mathcal{L}(y), \forall y \in \mathcal{S}$, with $\mathcal{S} \subset Y$ denoting a *training set*. These computations are done in an *offline* mode. Then, a *surrogate model* is constructed by designing a surrogate map, $\hat{\mathcal{L}} : Y \mapsto Z$

---

[*]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.

such that $\hat{\mathcal{L}}(y) \approx \mathcal{L}(y)$, for all $y \in \mathcal{S}$. Finally, in an *online* step, one evaluates $\hat{\mathcal{L}}(y)$ with $y \in Y \setminus \mathcal{S}$ to perform the prediction. This surrogate model will be effective as long as $\mathcal{L} \approx \hat{\mathcal{L}}$ in a suitable sense and the cost of evaluating the surrogate map $\hat{\mathcal{L}}$ is significantly lower than the cost of evaluating the underlying map $\mathcal{L}$. Examples of such surrogate models include reduced order models [34] and Gaussian process regression [37].

A particularly attractive class of such surrogate models are *deep neural networks* [15], i.e. functions formed by concatenated compositions of affine transformations and scalar non-linear activation functions. Deep neural networks have been extremely successful at diverse tasks in science and engineering [24] such as at image and text classification, computer vision, text and speech recognition, autonomous systems and robotics, game intelligence and even protein folding [11].

Deep neural networks are also being increasingly used in different contexts in scientific computing. A very incomplete sample of this rapidly growing field is in solutions of PDEs by so-called physics informed neural networks (PINNs) [23, 35, 36] and references therein, solutions of high-dimensional PDEs, particularly in finance [8, 9, 17] and improving the efficiency of existing numerical methods for PDEs, for instance in [41, 38, 28] and references therein.

Almost all of the afore-mentioned papers in scientific computing use deep neural networks in the context of *supervised learning* [15], i.e. training the tuning parameters (weights and biases) of the neural network to minimize the so-called loss function (difference between the underlying map and the neural network surrogate on the training set in some suitable norm) with a stochastic gradient descent method.

The accuracy of this supervised learning procedure is often expressed in terms of the so-called *generalization error* or population risk [39] (see (3.4) for the precise definition). The generalization error measures the error of the network on unseen data and it is notoriously hard to estimate it sharply [1, 29] and references therein. Nevertheless, it is customary to estimate it in terms of the following bound [39],

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

with $\mathcal{E}_G$ being the generalization error. Here, N is the number of training samples and $\mathcal{E}_T$ is the so-called *training error* or *empirical risk*, that is readily computed from the loss function (see (3.6) for a definition).

When the training samples are chosen randomly from the underlying probability distribution on Y, one can think of the upper bound U in (1.1) as a sort of *standard deviation* of the underlying map $\mathcal{L}$ (see section 2.3, eqn (2.22) of a recent paper [25] for a precise estimate and discussion of this issue). Even with this bound (1.1), one immediately confronts the following challenges for using neural networks as surrogates in this context:

- The bound (1.1) stems from an application of the central limit theorem and indicates a slow decay of the error with respect to the number of training samples. In particular, as long as the standard deviation of the underlying map $\mathcal{L}$ is $\mathcal{O}(1)$, we require a large number of training samples to obtain low generalization errors. For instance, $\mathcal{O}(10^4)$ training samples are needed in order to obtain a generalization error of 1% relative error. Computing such large number of training samples by evaluating the underlying PDEs is very expensive.

- Furthermore, the upper bound $U \sim \text{std}(\mathcal{L})$ only holds under the assumption of (statistical) independence of the evaluations of the trained neural network. This is not necessarily true as the training process (stochastic gradient descent) can lead to *strong correlations* between evaluations of the neural network, at different points of the training set. Estimating these correlations is the fundamental problem in obtaining sharp upper bounds on the generalization error [1, 10] and tools from statistical learning theory [5] such as the Vapnik-Chervonenkis (VC) dimension and Rademacher complexity [39] are often used for this bound. Consequently, the estimate (1.1) can indicate an even slower decay of the error.

As generating training data is expensive, the preceding discussion makes it clear that *learning* maps, at least in the context of scientific computing, with randomly chosen training samples can be hard. This is in contrast to the *big data* successes of deep learning in computer science where generating and accessing data (images, text, speech) is cheap.

One remedy for this problem would be to use *variance reduction procedures* to reduce the standard deviation of the underlying map in (1.1). A robust variance reduction method is the so-called multi-level

procedure [18, 13] and it was adapted to design a *multi-level training algorithm* for deep neural networks in the recent paper [25]. However, this multi-level training procedure, in conjunction with randomly chosen training points, is efficient only when the amplitude of correlations in the training process is low (see [25]) and might not be suitable for many problems.

Another possible solution to this problem of finding accurate neural network surrogates with a few training samples, was presented in a recent paper [26], where the authors proposed selecting *low-discrepancy sequences* as training sets for supervising learning with deep neural networks. Such low-discrepancy or *quasi-random* sequences were developed in the context of Quasi-Monte Carlo (QMC) methods for numerical integration [3, 6] and references therein and many variants of these are available. The key point in using low-discrepancy sequences for numerical integration is the fact that the resulting quadrature errors decay linearly with respect to the number of quadrature points (up to logarithmic corrections), at least as long the underlying integration is over a domain in moderately high dimensions, resulting in a clear gain over the competing Monte Carlo algorithms.

The heuristic argument presented in [26] to motivate the use of low-discrepancy sequences to train deep neural networks was based on the *equidistribution* property of these sequences, i.e. these sequences fill the underlying domain more uniformly than random points and hence can better represent the underlying map $\mathcal{L}$. Although the numerical results with such an algorithm as presented in [26], [25] (also for a multi-level version of this algorithm) were very convincing, no rigorous analysis was provided to explain the results and justify the underlying intuition.

The fundamental goal of the present paper is to describe and rigorously analyze this QMC type deep learning algorithm. To this end, in this paper we will

- Formalize and present a deep learning algorithm, based on choosing low-discrepancy sequences as training sets for supervised learning in deep neural networks.

- Analyze this algorithm by presenting upper bounds on the generalization error and using these upper bounds to constrain the choice of activation functions.

- Present a wide range of numerical experiments to illustrate the algorithm and the derived bounds as well as to demonstrate the advantage of the proposed algorithm over the standard supervised deep learning algorithm with randomly chosen training data.

Our aim would be to convince the reader of the efficacy of the proposed deep learning algorithm for a large range of applications in scientific computing, particularly for problems in moderately high dimensions.

The rest of the paper is organized as follows: in section 2, we present the proposed algorithm. The algorithm is analyzed and upper bounds on the generalization error are derived in section 3 and numerical experiments, illustrating the algorithm, are presented in section 4.

2 The deep learning algorithm

2.1 Problem setup

We consider an underlying domain $Y \subset \mathbb{R}^d$ and for definiteness we let $Y = [0,1]^d$ be the unit cube in d dimensions. Moreover, we also consider an underlying probability measure $\mu \in \mathcal{P}(Y)$ and further assume that this measure is absolutely continuous with respect to the d-dimensional Lebesgue measure and has a density given by $\bar{\mu} \in C(Y; \mathbb{R}_+)$ with

$$\int_Y \bar{\mu}(y)dy = 1.$$

Our objective is to find efficient surrogates for maps of the form,

$$\mathcal{L} : Y \mapsto \mathbb{R}^m. \tag{2.1}$$

Without loss of generality, we take $m = 1$.

A prototypical example for the map $\mathcal{L}$ comes from time-dependent *parametric PDEs* of the form,

$$\partial_t \mathbf{U}(t, x, y) = L\left(y, \mathbf{U}, \nabla_x \mathbf{U}, \nabla_x^2 \mathbf{U}, \ldots\right), \quad \forall (t, x, y) \in [0, T] \times D(y) \times Y,$$

$$\mathbf{U}(0, x, y) = \overline{\mathbf{U}}(x, y), \quad \forall (x, y) \in D(y) \times Y, \tag{2.2}$$

$$L_b \mathbf{U}(t, x, y) = \mathbf{U}_b(t, x, y), \quad \forall (t, x, y) \in [0, T] \times \partial D(y) \times Y.$$

Here, Y is the underlying parameter space. The spatial domain is labeled as $y \to D(y) \subset \mathbb{R}^{d_s}$ and $\mathbf{U} : [0, T] \times D \times Y \to \mathbb{R}^M$ is the vector of unknowns. The differential operator L is in a very generic form and can depend on the gradient and Hessian of $\mathbf{U}$, and possibly higher-order spatial derivatives. For instance, the heat equation as well as the Euler or Navier-Stokes equations of fluid dynamics are specific examples. Moreover, L_b is a generic operator for imposing boundary conditions. The parametric nature of the PDE (2.2), represented by the parameter space Y, can stem from uncertainty quantification or Bayesian inversion problems where the parameter space models uncertainties in inputs for the underlying PDE. The parametric nature can also arise from optimal design, control and PDE constrained optimization problems with Y being the design (control) space.

For the parameterized PDE (2.2), we consider the following generic form of *observables*,

$$L_g(y, \mathbf{U}) := \int_0^T \int_{D(y)} \psi(x, t) g(\mathbf{U}(t, x, y)) dx \, dt, \quad \text{for } \mu \text{ a.e } y \in Y. \tag{2.3}$$

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

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

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

with L_g being defined by (2.3). In general, it is only possible to evaluate $\mathcal{L}$ up to numerical error, by approximating it with a suitable numerical method.

We will design a deep learning algorithm for approximating the underlying map $\mathcal{L}$ (2.1) in the following steps.

2.2 Training set

As is customary in supervised learning ([15] and references therein), we need to generate or obtain data to train the network. To this end, we fix $N \in \mathbb{N}$ and select a set of points $\mathcal{S} = \{y_i\}_{1 \leqslant i \leqslant N}$, with each $y_i \in Y$. It is standard that the points in the training set $\mathcal{S}$ are chosen randomly from the parameter space Y, independently and identically distributed with the measure μ. However, in this paper we follow [26] and choose *low-discrepancy sequences* for our training set.

2.3 Low-discrepancy sequences

Consider any sequence of points $\{y_n\}_{1 \leqslant n \leqslant N}$ with $y_n \in Y = [0, 1]^d$ for all n, and for any set $J \subset Y$ denote

$$R_N(J) := \frac{1}{N} \# \{y_n \in J\} - \text{meas}(J). \tag{2.5}$$

Note that (2.5) is the error of approximating the Lebesgue measure meas of the set $J \subset Y$ with the sum $\frac{1}{N} \sum_{n=1}^{N} \chi_J(y_n)$, with χ denoting the indicator function of the underlying set.

Instead of considering arbitrary subsets of Y, we focus on *rectangular sets* [3]. To define a d-dimensional rectangle, we need only two antipodal vertices. In particular, we consider the following rectangular sets,

$$E^* := \{J \subset Y : J \text{ is a rectangular set with one vertex } 0 \text{ another vertex } z \in Y\}.$$

Then, we follow [3] and define the star-discrepancy or simply, discrepancy, of the sequence $\{y_n\}$ as,

$$D_N^* = \sup_{J \in E^*} |R_N(J)|. \tag{2.6}$$

4

Roughly speaking, the discrepancy measures how well the sequence fills the underlying domain and consequently, how well is the measure of an arbitrary rectangular subset J of Y approximated by the sum $\frac{1}{N} \sum_{n=1}^{N} \chi_J(y_n)$. We are now in a position to follow [3] and define low-discrepancy sequences below.

Definition 2.1. Low-discrepancy sequence: *A sequence of points $\{y_n\}_{1 \leqslant n \leqslant N}$ with $y_n \in Y$ is termed as a low-discrepancy sequence if the following holds for its discrepancy D_N^* (2.6):*

$$D_N^* \leqslant C \frac{(\log N)^d}{N}, \tag{2.7}$$

with a constant C that possibly depends on the dimension d but is independent of N.

It turns out that there are different types of low-discrepancy sequences, many of them are built from number theoretic considerations. The simplest low-discrepancy sequence is the one-dimensional van der corput sequence. Widely used low-discrepancy sequences (in arbitrary dimensions) are the Sobol [40], Halton [16], Owen [31] and Niederreiter [30] sequences.

It is also straightforward to see that for a sequence of randomly chosen points, we have

$$D_N^* \sim O\left(\frac{1}{\sqrt{N}}\right). \tag{2.8}$$

Thus, randomly chosen points do not constitute a low-discrepancy sequence as defined in 2.1. This provides a formalism to the intuition that a sequence of low-discrepancy points fill the underlying space more uniformly than random points.

Once the training set $\mathcal{S}$ is chosen, we perform a set of simulations to obtain $\mathcal{L}(y)$, for all $y \in \mathcal{S}$. This might involve expensive simulations of the underlying PDE (2.2).
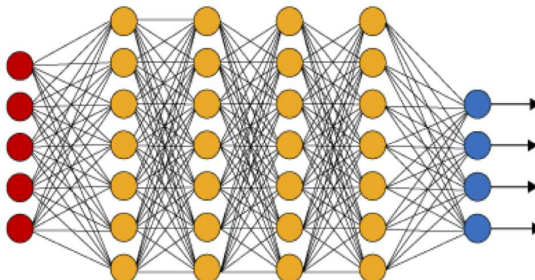

Figure 1: An illustration of a (fully connected) deep neural network. The red neurons represent the inputs to the network and the blue neurons denote the output layer. They are connected by hidden layers with yellow neurons. Each hidden unit (neuron) is connected by affine linear maps between units in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

2.4 Neural network

Given an input vector $y \in Y$, a feedforward neural network (also termed as a multi-layer perceptron), shown in figure 1, transforms it to an output through layers of units (neurons) consisting of either affine-linear maps between units (in successive layers) or scalar non-linear activation functions within units [15], resulting in the representation,

$$\mathcal{L}_\theta(y) = C_K \circ \sigma \circ C_{K-1} \ldots \ldots \ldots \circ \sigma \circ C_2 \circ \sigma \circ C_1(y). \tag{2.9}$$

Here, $\circ$ refers to the composition of functions and σ is a scalar (non-linear) activation function. A large variety of activation functions have been considered in the machine learning literature [15]. Popular choices for the activation function σ in (2.9) include the sigmoid function, the tanh function and the *ReLU* function defined by,

$$\sigma(z) = \max(z, 0). \tag{2.10}$$

5

When, $z \in \mathbb{R}^p$ for some $p > 1$, then the output of the ReLU function in (2.10) is evaluated componentwise. For any $1 \leqslant k \leqslant K$, we define

$$C_k z_k = W_k z_k + b_k, \quad \text{for } W_k \in \mathbb{R}^{d_{k+1} \times d_k}, z_k \in \mathbb{R}^{d_k}, b_k \in \mathbb{R}^{d_{k+1}}. \tag{2.11}$$

For consistency of notation, we set $d_1 = d$ and $d_{K+1} = 1$.

Thus in the terminology of machine learning (see also figure 1), our neural network (2.9) 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+1} neurons) is given an input vector $z_k \in \mathbb{R}^{d_k}$ and transforms it first by an affine linear map C_k (2.11) and then by a ReLU (or another) nonlinear (component wise) activation σ (2.10). A straightforward addition shows that our network contains $\left(d + 1 + \sum_{k=2}^{K} d_k \right)$ neurons. We also denote,

$$\theta = \{W_k, b_k\}, \theta_W = \{W_k\} \quad \forall \, 1 \leqslant k \leqslant K, \tag{2.12}$$

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

$$M = \sum_{k=1}^{K} (d_k + 1) d_{k+1}. \tag{2.13}$$

2.5 Loss functions and optimization

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

$$J(\theta) := \sum_{y \in \mathcal{S}} |\mathcal{L}(y) - \mathcal{L}_\theta(y)|^p \bar{\mu}(y), \tag{2.14}$$

for some $1 \leqslant p < \infty$ and with $\bar{\mu}$ being the density of the underlying probability distribution μ.

The goal of the training process in machine learning is to find the weight vector $\theta \in \Theta$, for which the loss function (2.14) is minimized.

It is common in machine learning [15] to regularize the minimization problem for the loss function, i.e. we seek to find

$$\theta^* = \arg\min_{\theta \in \Theta} \left(J(\theta) + \lambda \mathcal{R}(\theta) \right). \tag{2.15}$$

Here, $\mathcal{R} : \Theta \to \mathbb{R}$ is a *regularization* (penalization) term. A popular choice is to set $\mathcal{R}(\theta) = \|\theta_W\|_q^q$ for either $q = 1$ (to induce sparsity) or $q = 2$. The parameter $0 \leqslant \lambda \ll 1$ balances the regularization term with the actual loss J (2.14).

The above minimization problem amounts to finding a minimum of a possibly non-convex function over a subset of $\mathbb{R}^M$ for possibly very large M. We follow standard practice in machine learning by either (approximately) solving (2.15) with a full-batch gradient descent algorithm or variants of mini-batch stochastic gradient descent (SGD) algorithms such as ADAM [21].

For notational simplicity, we denote the (approximate, local) minimum weight vector in (2.15) as θ^* and the underlying deep neural network $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ will be our neural network surrogate for the underlying map $\mathcal{L}$.

The proposed algorithm for computing this neural network is summarized below.

Algorithm 2.2. Deep learning of parameters to observable map

Inputs: *Underlying map $\mathcal{L}$ (2.1) and low-discrepancy sequences in Y.*

Goal: *Find neural network $\mathcal{L}_{\theta^*}$ for approximating the underlying map $\mathcal{L}$.*

Step 1: *Choose the training set $\mathcal{S} = \{y_n\}$ for $y_n \in Y$, for all $1 \leqslant n \leqslant N$ such that the sequence $\{y_n\}$ is a low-discrepancy sequence in the sense of definition 2.1. Evaluate $\mathcal{L}(y)$ for all $y \in \mathcal{S}$ by a suitable numerical method.*

Step 2: *For an initial value of the weight vector $\overline{\theta} \in \Theta$, evaluate the neural network $\mathcal{L}_{\overline{\theta}}$ (2.9), the loss function (2.15) and its gradients to initialize the (stochastic) gradient descent algorithm.*

Step 3: *Run a stochastic gradient descent algorithm till an approximate local minimum θ^* of (2.15) is reached. The map $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ is the desired neural network approximating the map $\mathcal{L}$.*

3 Analysis of the deep learning algorithm 2.2

In this section, we will rigorously analyze the deep learning algorithm 2.2 and derive bounds on its accuracy. To do so, we need the following preliminary notions of variation of functions.

3.1 On Variation of functions

For ascertaining the accuracy of the deep learning algorithm 2.2, we need some hypotheses on the underlying map $\mathcal{L}$. As is customary in the analysis of Quasi-Monte Carlo (QMC) methods for numerical integration of functions [3], we will express the regularity of the underlying map $\mathcal{L}$ in terms of its *variation*. As long as $d = 1$, the unambiguous notion of variation is given by the total variation of the underlying function. However, there are multiple notions of variations for multi-variate functions. In this section, we very closely follow the presentation of [32] to consider variation in the sense of Vitali and Hardy-Krause.

To this end, we consider scalar valued functions defined on a hyperrectangle $[\mathbf{a}, \mathbf{b}] = \{x \in \mathbb{R}^d \mid a_i \leqslant x_i \leqslant b_i \ \forall i = 1, \ldots, d\}$ and define ladders $\mathcal{Y} = \prod_{j=1}^d \mathcal{Y}^j$ on $[\mathbf{a}, \mathbf{b}]$, where $\mathcal{Y}^j$ for each $j = 1, \ldots, d$ is a set of finitely many values in (a_j, b_j). Each $y^j \in \mathcal{Y}^j$ has a successor y_+^j, which is the smallest element in $(y^j, \infty) \cap \mathcal{Y}^j$ if $(y^j, \infty) \cap \mathcal{Y}^j \neq \emptyset$, and b_j otherwise. Consequently, we define $\mathbf{y}_+$ to be the successor of $\mathbf{y} \in \mathcal{Y}$, where $(\mathbf{y}_+)_j$ is the succesor of $(\mathbf{y})_j$ for all $j = 1, \ldots, d$. Furthermore, we define $\mathbb{Y}$ to be the set of all ladders on $[\mathbf{a}, \mathbf{b}]$.

Definition 3.1. *The variation of a function f on the hyperrectangle $[\mathbf{a}, \mathbf{b}]$ in the sense of Vitali is given as*

$$V(f) := V_{[\mathbf{a},\mathbf{b}]}(f) = \sup_{\mathcal{Y} \in \mathbb{Y}} \sum_{\mathbf{y} \in \mathcal{Y}} \sum_{v \subset 1:d} (-1)^{|v|} f(\mathbf{y}^v : \mathbf{y}_+^{-v}),$$

where $\mathbf{y}^v : \mathbf{y}_+^{-v}$ is the glued vector $\hat{\mathbf{y}} \in [\mathbf{a}, \mathbf{b}]$ with

$$\hat{y}_i = \begin{cases} y_i & i \in v \\ (y_+)_i & i \notin v \end{cases}.$$

Hence, for $\mathbf{x} \in [\mathbf{a}, \mathbf{b}]$ the object $\mathbf{x}^{-v}$ denotes the complement with respect to $1 : d$ for all $v \subset 1 : d$.

The Hardy-Krause variation can now be defined based on the Vitali Variation.

Definition 3.2. *The variation of a function f on the hyperrectangle $[\mathbf{a}, \mathbf{b}]$ in the sense of Hardy-Krause is given as*

$$V_{HK}(f) = \sum_{u \subsetneq 1:d} V_{[\mathbf{a}^{-u}, \mathbf{b}^{-u}]}(f(\mathbf{x}^{-u}; \mathbf{b}^u)), \tag{3.1}$$

where $f(\mathbf{x}^{-u}; \mathbf{b}^u)$ denotes the function $f(\mathbf{x}^{-u} : \mathbf{b}^u)$ with $\mathbf{x}^{-u}$ is the argument and b^u is a parameter.

Following this definition it is clear that $V(f) < \infty$ must hold in order for $V_{HK}(f) < \infty$ to hold. Additionally, V_{HK} is a semi-norm, as it vanishes for constant functions. However, it can be extended to a full norm by adjoining the case $u = 1 : d$ to the sum in (3.1).

For notational simplicity, we will fix the hyperrectangle $[\mathbf{a}, \mathbf{b}]$ to be the unit cube $Y = [0, 1]^d$ in $\mathbb{R}^d$.

Remark 3.3. Given a function f, it is clearly impractical to compute its Hardy-Krause variation in terms of formula (3.1). On the other hand, as long as the underlying function f is smooth enough, we can use the following straightforward bound for the Hardy-Krause variation [32]:

$$V_{HK}(f) \leqslant \hat{V}_{HK} = \int_Y \left| \frac{\partial^d f(y)}{\partial y_1 \partial y_2 \cdots \partial y_d} \right| dy + \sum_{i=1}^d \hat{V}_{HK}(f_1^{(i)}). \tag{3.2}$$

Here, $f_1^{(i)}$ is the restriction of the function f to the boundary $y_i = 1$. Since this restriction is a function in $d - 1$ dimensions, the above formula (3.2) is understood as a recursion with the Hardy-Krause variation of a univariate function being given by its total variation:

$$\hat{V}_{HK}(f) = \int_0^1 \left| \frac{df}{dy}(y)dy \right|. \tag{3.3}$$

Note that the inequality in (3.2) is an identity as long as all the mixed partial derivatives that appear in (3.2) are continuous [3].

■

3.2   Estimates on the generalization error of algorithm 2.2

We fix $p = 1$ in the loss function (2.14). Our aim in this section is to derive bounds on the so-called *generalization error* [39] of the trained neural network $\mathcal{L}^*$ (2.9) generated by the deep learning algorithm 2.2, which is customarily defined by,

$$\mathcal{E}_G = \mathcal{E}_G(\theta^*; \mathcal{S}) := \int_Y |\mathcal{L}(y) - \mathcal{L}^*(y)| \bar{\mu}(y) dy. \tag{3.4}$$

Note that this generalization error depends explicitly on the training set $\mathcal{S}$ and on the parameters θ^* of the trained neural network as $\mathcal{L}^*$ depends on them. However, we will suppress this dependence for notational convenience. Also, for notational convenience, we will assume that the underlying probability distribution μ is *uniform*, i.e. $\bar{\mu} \equiv 1$. All the estimates presented below can be readily but tediously extended to the case of $\bar{\mu} \in C^d(Y)$ and we omit these derivations here.

Thus, we consider the generalization error of the form:

$$\mathcal{E}_G = \mathcal{E}_G(\theta^*; \mathcal{S}) := \int_Y |\mathcal{L}(y) - \mathcal{L}^*(y)| dy. \tag{3.5}$$

For each training set $\mathcal{S} = \{y_n\}$ with $1 \leq n \leq N$, the training process in algorithm 2.2 amounts to minimize the so-called *training error*:

$$\mathcal{E}_T = \mathcal{E}_T(\theta^*; \mathcal{S}) := \frac{1}{N} \sum_{i=1}^N |\mathcal{L}(y_n) - \mathcal{L}^*(y_n)|. \tag{3.6}$$

The training error can be calculated from the loss function (2.14), a posteriori after the training is concluded. Note that the training error explicitly depends on the training set $\mathcal{S}$ and we suppress this dependence.

Given these definitions, we wish to estimate the generalization error in terms of the computable training error. This amounts to estimating the so-called *generalization gap*, i.e. the difference between the generalization and training error. We do so in the lemma below.

Lemma 3.4. *Let $\mathcal{L}^*$ be a deep neural network of the form (2.9), generated by the deep learning algorithm 2.2, with the training set $\mathcal{S} = \{y_n\}$, for $y_n \in Y$ with $1 \leq n \leq N$ being a low-discrepancy sequence in the sense of (2.7). Then the generalization gap is estimated by,*

$$|\mathcal{E}_G - \mathcal{E}_T| \leq C \frac{V_{HK}(|\mathcal{L} - \mathcal{L}^*|)(\log N)^d}{N}, \tag{3.7}$$

with V_{HK} being the Hardy-Krause variation defined in (3.1).

By observing the definitions of the generalization error (3.5) and training error (3.6), we see that the training error $\mathcal{E}_T$ is exactly the Quasi-Monte Carlo approximation of the generalization error $\mathcal{E}_G$, with the

underlying integrand being $|\mathcal{L} - \mathcal{L}^*|$. Therefore, we can apply the well-known Koksma-Hlawka inequality [3] to obtain,

$$|\mathcal{E}_G - \mathcal{E}_T| \leqslant V_{HK}\left(|\mathcal{L} - \mathcal{L}^*|\right) D_N^* \quad \text{(by Koksma − Hlawka inequality)}$$

$$\leqslant C V_{HK}\left(|\mathcal{L} - \mathcal{L}^*|\right) \frac{(\log N)^d}{N}, \quad \text{(by (2.7))}$$

which is the desired inequality (3.7).

Thus, we need to assume that the Hardy-Krause variation of the underlying map $\mathcal{L}$ is bounded as well as show that the Hardy-Krause variation of the trained neural network is finite in order to bound the generalization error. We do so in the lemma below.

Lemma 3.5. *Assume that the underlying map $\mathcal{L} : Y \mapsto \mathbb{R}$ is such that*

$$\hat{V}_{HK}(\mathcal{L}) < \infty. \tag{3.8}$$

Assume that the neural network $\mathcal{L}^*$ is of the form (2.9) with an activation function $\sigma : \mathbb{R} \mapsto \mathbb{R}$ with $\sigma \in C^d(\mathbb{R})$. Then, for any given tolerance $\delta > 0$, the generalization error (3.5) with respect to the neural network $\mathcal{L}^*$, generated by the deep learning algorithm 2.2, can be estimated as,

$$\mathcal{E}_G \leqslant \mathcal{E}_T + \frac{C(\log N)^d}{N} + 2\delta. \tag{3.9}$$

Here, the constant C depends on the tolerance δ but is independent of the number of training samples N.

Proof. To prove (3.9), for any given $\delta > 0$, we use the following auxiliary function,

$$\rho_\delta \in C^d(\mathbb{R} \mapsto \mathbb{R}_+) : \; \||u| - \rho_\delta(u)\|_{L^\infty(\mathbb{R})} \leqslant \delta. \tag{3.10}$$

The existence of such a function ρ_δ can easily be verified by mollifying the absolute value function in the neighborhood of the origin.

We denote,

$$\mathcal{E}_G^\delta := \int_Y \rho_\delta\left(\mathcal{L}(y) - \mathcal{L}^*(y)\right) dy, \quad \mathcal{E}_T^\delta = \frac{1}{N}\sum_{i=1}^N \rho_\delta\left(\mathcal{L}(y_n) - \mathcal{L}^*(y_n)\right).$$

Then, it is easy to see that $\mathcal{E}_T^\delta$ is the QMC approximation of $\mathcal{E}_G^\delta$ and we can apply the Koksma-Hlawska inequality as in the proof of lemma 3.4 to obtain that,

$$|\mathcal{E}_G^\delta - \mathcal{E}_T^\delta| \leqslant C\frac{V_{HK}\left(\rho_\delta\left(\mathcal{L} - \mathcal{L}^*\right)\right)(\log N)^d}{N}. \tag{3.11}$$

From the assumption that the activation function $\sigma \in C^d(\mathbb{R})$ and structure of the artificial neural network (2.9), we see that for any $\theta \in \Theta$, the neural network $\mathcal{L}^*$ is a composition of affine (hence C^d) and sufficiently smooth C^d functions, with each function in the composition being defined on compact subsets of $\mathbb{R}^{\bar{d}}$, for some $\bar{d} \geqslant 1$. Thus, for this composition of functions, we can directly apply Theorem 4 of [2] to conclude that

$$V_{HK}(\mathcal{L}^*) \equiv \hat{V}_{HK}(\mathcal{L}^*) < \infty.$$

Moreover, we have assumed that $\hat{V}_{HK}(\mathcal{L}) < \infty$. It is straightforward to check using the definition (3.2) that

$$\hat{V}_{HK}(\mathcal{L} - \mathcal{L}^*) < \infty. \tag{3.12}$$

Next, we observe that $\rho_\delta \in C^d(\mathbb{R})$ by our assumption (3.10). Combining this with (3.12), we use Theorem 4 of [2] (with an underlying multiplicative Faa di Bruno formula for compositions of multi-variate functions) to conclude that

$$V_{HK}\left(\rho_\delta\left(\mathcal{L} - \mathcal{L}^*\right)\right) \leqslant C < \infty. \tag{3.13}$$

Here, the constant C depends on the dimension and the tolerance δ.

Applying (3.13) in (3.11) and identifying all the constants as C yields,

$$|\mathcal{E}_G^\delta - \mathcal{E}_T^\delta| \leqslant C(d,\delta)\frac{(\log N)^d}{N}. \tag{3.14}$$

It is straightforward to conclude from assumption (3.10) that

$$\max\left\{|\mathcal{E}_G - \mathcal{E}_G^\delta|, |\mathcal{E}_T - \mathcal{E}_T^\delta|\right\} \leqslant \delta. \tag{3.15}$$

Combining (3.14) with (3.15) and a straightforward application of the triangle inequality leads to the desired inequality (3.9) on the generalization error. $\qquad\square$

Several remarks about Lemma 3.5 are in order.

Remark 3.6. The estimate (3.9) requires a certain regularity on the underlying map $\mathcal{L}$, namely that $\hat{V}_{HK}(\mathcal{L}) < \infty$. This necessarily holds if the underlying map $\mathcal{L} \in C^d$. However, this requirement is significantly weaker than assuming that the underlying map is d-times continuously differentiable. Note that the condition of the Hardy-Krause variation is only on the mixed-partial derivatives and does not require any boundedness on other partial derivatives. Moreover, in many applications, for instance when the underlying parametric PDE (2.2) is elliptic or parabolic and the functions ψ, g in (2.3) are smooth, this regularity is always observed. $\qquad\blacksquare$

Remark 3.7. It is instructive at this stage to compare the deep learning algorithm 2.2 based on a low-discrepancy sequence training set with one based on randomly chosen training sets. As mentioned in the introduction, the best-case bound on the generalization error for randomly chosen training sets is given by (1.1) with bound,

$$\mathcal{E}_G \sim \mathcal{E}_T + \frac{\text{std}(\mathcal{L})}{\sqrt{N}}, \tag{3.16}$$

with the above inequality holding in the root mean square sense (see estimate (2.22) and the discussion around it in the recent article [25]). Comparing such an estimate to (3.9) leads to the following observations:

- First, we see from (3.9), that as long as the number of training samples $N \geqslant 2^d$, the generalization error for the deep learning algorithm with a low-discrepancy sequence training set decays at much faster (linear) rate than the corresponding $1/\sqrt{N}$ decay for the deep learning algorithm with a randomly chosen training set.

- As is well known [39, 10, 1], the bound (3.16) on the generalization error needs not necessarily hold due to a high degree of correlations in the trained neural network when evaluated on training samples. Consequently, the bound (3.16), and the consequent rate of convergence of the deep learning algorithm with randomly chosen training set with respect to number of training samples can be significantly worse. No such issue arises for the bound (3.9) on the generalization error for the deep learning algorithm 2.2 with a low-discrepancy sequence training set, as this bound is based on the Koksma-Hlawka inequality and automatically takes correlations into account.

Given the above considerations, we can expect that at least till the dimension of the underlying map is moderately high and if the map is sufficiently regular, the deep learning algorithm 2.2 with a low-discrepancy sequence training set will be *significantly more accurate* than the standard deep learning algorithm with a randomly chosen training set. On the other hand, either for maps with low-regularity or for problems in very high dimensions, the theory developed here suggests no significant advantage using a low-discrepancy sequence as the training set over randomly chosen points, in the context of deep learning. $\qquad\blacksquare$

Remark 3.8. In principle, the upper bound (3.9) on the generalization error is computable as long as good estimates on $\hat{V}_{HK}(\mathcal{L})$ are available. In particular, the training error $\mathcal{E}_T$ (3.6) is computed during the training process. The constants $C = C(d,\delta)$ in (3.9) can be explicitly computed from the generalized Faa di Bruno formula (eqn [10] in reference [2]) and given the explicit form (2.9) of the neural network, we can compute $\hat{V}_{HK}(\mathcal{L}^*)$ explicitly by computing the mixed partial derivatives. However, it is well known that the estimates on QMC integration error in terms of the Koksma-Hlawka inequality are very large

overestimates [3]. Given this, we suspect that bounds of the form (3.9) will be severe overestimates. On the other hand, the role of the bound (3.9) is to demonstrate the linear decay of the error, with respect to the number of training samples, at least up to moderately high dimensions. ∎

Remark 3.9. Finally, we investigate the issue of activation functions in this context. In deriving the bound (3.9), we assume that the activation function σ in the neural network (2.9) is sufficiently smooth, i.e. $\sigma \in C^d(\mathbb{R})$. Thus, standard choices of activation functions such as sigmoid and tanh are admissible under this assumption.

On the other hand, the ReLU function (2.10) is widely used in deep learning. Can we use ReLU also as an activation function in the context of the deep learning algorithm 2.2, with low-discrepancy sequences as training sets? A priori, the generalization error bound (3.7) only requires that $|\mathcal{L} - \mathcal{L}^*|$ has bounded Hardy-Krause variation (3.1). Note that this expression does not require the existence of mixed partial derivatives, up to order d, of the neural network $\mathcal{L}^*$ and hence, of the underlying activation function. Thus, ReLU might very well be an admissible activation function in this context.

However, in [32], the author considers the following function for any non-negative integers d, r:

$$f_{d,r}(x) = \left(\max \left\{ \sum_{i=1}^{d} x_i - \frac{1}{2}, 0 \right\} \right)^r, \tag{3.17}$$

and proves using rather elementary arguments with a ladder $\mathcal{Y}_i = \{0, 1/(2m), \ldots, (m-1)/(2m)\}$, that as long as $d \geqslant r + 2$, the function $f_{d,r}$ (3.17) has infinite variation in the sense of Vitali, and hence infinite Hardy-Krause variation (3.1).

This argument can be extended in the following: for $d \geqslant 3$, consider the following simple 2-layer ReLU network $\mathcal{L}_d^* : Y \subset \mathbb{R}^d \mapsto \mathbb{R}_+$:

$$\mathcal{L}_d^*(y) = \max \left\{ \sum_{i=1}^{d} y_i - \frac{1}{2}, 0 \right\}. \tag{3.18}$$

Note that (3.18) is a neural network of the form (2.9) with a single hidden layer, ReLU activation function, all weights being 1 and bias is -0.5. We conclude from [32] (proposition 16) that the Hardy-Krause variation of the ReLU neural network (3.18) is infinite. Thus, using ReLU activation functions in (2.9) might lead to a blow-up of the upper bound in the generalization error (3.7) and a reduced rate of convergence of the error with respect to the number of training samples.

However, it is worth pointing out that the bound (3.7) can be a gross over-estimate and in practice, we might well have much better generalization error with ReLU activation functions than what (3.7) suggests. To test this proposition, we consider the following simple numerical example. We let $Y = [0, 1]^3$, i.e. Y is a 3-dimensional cube. The underlying map is given by $\mathcal{L}(y) = f_{3,4}(y)$, with $f_{3,4}$ defined from (3.17). Clearly, for this underlying map $\hat{V}_{HK}(\mathcal{L}) < \infty$. Thus, using a sigmoid activation function in (2.9) and applying the generalization error estimate (3.9) suggests a linear decay of the generalization error. On the other hand, (3.9) does not directly apply if the underlying activation function is ReLU (2.10). We approximate this underlying map $\mathcal{L}(y) = f_{3,4}(y)$ with the following three algorithms:

- The deep learning algorithm 2.2 with Sobol sequences as the training set and sigmoid activation function. This configuration is referred to as DL_{sob}.

- The deep learning algorithm 2.2 with Sobol sequences as the training set and ReLU activation function. This configuration is referred to as ReLU-DL_{sob}.

- The deep learning algorithm 2.2 with Random points as the training set and sigmoid activation function. This configuration is referred to as DL_{rand}.

For each configuration, we train deep neural networks with architectures and hyperparameters specified in table 1 with exactly 1 hidden layer. We train 100 of these one-hidden layer networks and present the average (over the ensemble) generalization error (approximated on a test set with 8192 sobol points) for each of the above 3 configurations, for different numbers of training points, and present the results in figure 2. As seen from the figure, the generalization error with the DL_{sob} algorithm is significantly smaller than the other 2 configurations and decays superlinearly with respect to the number of training samples.

On the other hand, there is no significant difference between the ReLU-DL_{sob} and DL_{rand} algorithms for this example. The generalization error decays much more slowly than for the DL_{sob} algorithm. Thus, this example illustrates that it might not be advisable to use ReLU as an activation function in the deep learning algorithm 2.2 with low-discrepancy sequences. Smooth activiation functions, such as sigmoid and tanh should be used instead. ∎

4 Numerical experiments

4.1 Details of implementation

We have implemented the deep learning algorithm 2.2 in two versions: the DL_{sob} algorithm, with a Sobol sequence [40] as the training set and the DL_{rand} algorithm with a randomly chosen training set. Both versions are based on networks with sigmoid activation functions. The test set for each of the experiments below has a size of 8192 sampling points, which sufficiently outnumbers the number of training samples, and is based on the Sobol sequence when testing the DL_{sob} algorithm and are uniformly random drawn when testing the DL_{rand} algorithm.

There are several hyperparameters to be chosen in order to specify the neural network (2.9). We will perform an ensemble training, on the lines proposed in [26], to ascertain the best hyperparameters. The ensemble of hyperparameters is listed in table 1. The weights of the networks are initialized according to the Xavier initialization [14], which is standard for training networks with sigmoid (or tanh) activation functions. We also note that we choose $p = 2$ in the loss function (2.14) and add the widely-used weight decay regularization to our training procedure, which is based on an L^2-penalization of the weights in (2.15). Although the theory has been developed in the L^1-setting for low-discrepancy sequences, we did not find any significant difference between the results for $p = 1$ or $p = 2$ in the training loss function.

If not stated otherwise, for all experiments, we present the generalization error (by approximating it with an error on a test set) for the best performing network, selected in the ensemble training, for the DL_{sob} algorithm. On the other hand, as Monte Carlo methods are inherently statistical, the average over 100 retrainings (random initializations of weights and biases) of the test error will be presented for the DL_{rand}-algorithm.

The training is performed with the well-known ADAM optimizer [21] in full-batch mode for a maximum of 20000 epochs (or training steps).

The experiments are implemented in the programming language Python using the open source machine learning framework PyTorch [33] to represent and train the neural network models. The scripts to perform the ensemble training as well as the retraining procedure together with all data sets used in the experiments can be downloaded from *https://github.com/tk-rusch/Deep_QMC_sampling*. Next, we present a set of numerical experiments that are selected to represent different areas in scientific computing.

Table 1: Hyperparameters for the ensemble training

Hyperparameter	values used for the ensemble
learning rate	$10^{-1}, 10^{-2}, 10^{-3}$
weight decay	$10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}$
depth	$2^2, 2^3, 2^4$
width	$3 \times 2^1, 3 \times 2^2, 3 \times 2^3$

4.2 Function approximation: sum of sines

We start with an experiment that was proposed in [26] to test the abilities of neural networks to approximate nonlinear maps in moderately high dimension. The map to be approximated is the following sum of sines:

$$\mathcal{L}(x) = \sum_{i=1}^{6} \sin(4\pi x_i), \tag{4.1}$$

where $x \in [0,1]^6$. Clearly $\mathcal{L}$ is infinitely differentiable and satisfies the assumptions of Lemma 3.5. On the other hand, given that we are in 6 dimensions and that the derivatives of $\mathcal{L}$ are large, it is quite challenging to approximate this map by neural networks [26].

In fact the experiments in [26] show that the standard DL_{rand} algorithm produces large errors even for a very large number of training samples due to a high training error.

Motivated by this, we modify the deep learning algorithm slightly by introducing the well-known practice of *batch normalization* [20] to each hidden layer of the neural network. Using this modification of our network architectures, the training errors with the resulting algorithm, plotted in figure 3, are now sufficiently small. Moreover, the generalization error for the DL_{rand} algorithm decays at a rate of 0.63 with respect to the number of training samples. This is slightly better than the predicted rate of 0.5.
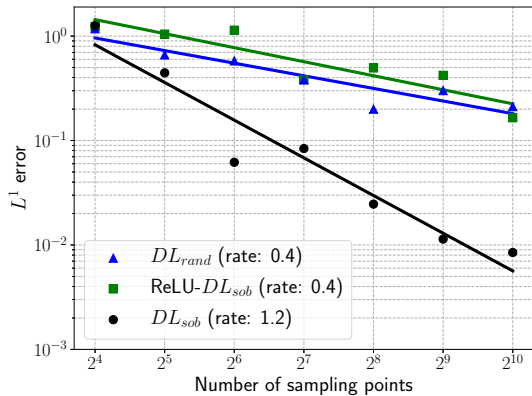
Figure 2: L^1 generalization errors for approximating $f_{3,4}$ (3.17) using DL_{sob} based on ReLU and sigmoid activation functions compared to the generalization error when using DL_{rand}.
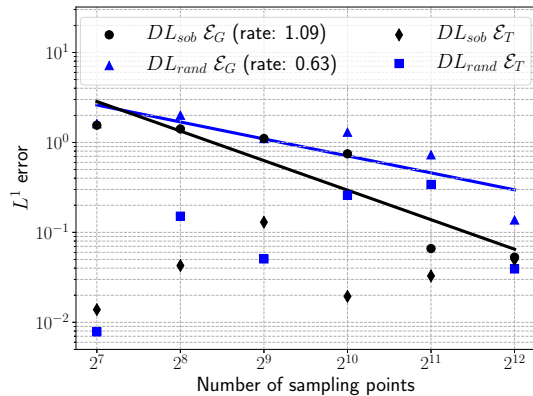
Figure 3: L^1 generalization and training errors for approximating the sum of sines (4.1) underlying mapping using DL_{sob} as well as DL_{rand}.

On the other hand, the DL_{sob} algorithm, based on low-discrepancy Sobol points does much better. Indeed, the training error with DL_{sob} is lower than the training error with the DL_{rand} algorithm with most training sets, but not in all of them. However, the generalization gap (and hence the generalization error) of DL_{sob} is significantly smaller than that of DL_{rand} and decays at the rate of 1.1, which is slightly better than the expected rate of 1. This experiment clearly validates the theory developed earlier in the paper. Indeed comparing (3.16) with (3.9) shows that even if the differences in training errors might be minor (we have no control on the size of this error in (3.9)), the generalization gap with the DL_{sob} algorithm should be significantly smaller than the DL_{rand} algorithm. This is indeed observed in the numerical results presented in figure 3.

4.3   UQ for ODEs: projectile motion.

This problem is from [25] where it was introduced as a prototype for learning observables for ODEs and dynamical systems. The motion of a projectile, subjected to gravity and air drag, is described by the following system of ODEs,

$$\frac{d}{dt}\mathbf{x}(t;y) = \mathbf{v}(t;y), \qquad\qquad \mathbf{x}(0;y) = \mathbf{x}_0(y), \qquad\qquad (4.2)$$

$$\frac{d}{dt}\mathbf{v}(t;y) = -F_D(\mathbf{v}(t;y);y)\mathbf{e}_1 - g\mathbf{e}_2, \qquad \mathbf{v}(0;y) = \mathbf{v}_0(y), \qquad\qquad (4.3)$$

where $F_D = \frac{1}{2m}\rho C_d \pi r^2 \|v\|^2$ denotes the drag force. Additionally, the initial conditions are set to $\mathbf{x}(0;y) = [0,h]$ and $\mathbf{v}(0;y) = [v_0\cos(\alpha), v_0\sin(\alpha)]$.

In the context of uncertainty quantification (UQ) [12], one models uncertainty in the system on account

of measurement errors, in a statistical manner, leading to a parametric model of the form,

$$\rho(y) = 1.225(1 + \varepsilon G_1(y)), \ r(y) = 0.23(1 + \varepsilon G_2(y)), \ C_D(y) = 0.1(1 + \varepsilon G_3(y)),$$

$$m(y) = 0.145(1 + \varepsilon G_4(y)), \ h(y) = (1 + \varepsilon G_5(y)), \ \alpha(y) = 30(1 + \varepsilon G_6(y)), \ v_0(y) = 25(1 + \varepsilon G_7(y)),$$

where $y \in Y = [0,1]^7$ is the parameter space, with an underlying uniform distribution. Additionally, $G_k(y) = 2y_k - 1$ for all $k = 1, \ldots, 7$ and $\varepsilon = 0.1$.

We choose the *horizontal range* $x_{\max}$ to be the observable of the simulation:

$$x_{\max}(y) = x_1(y, t_f), \qquad \text{with } t_f = x_2^{-1}(0).$$

The objective is to predict and approximate the map $\mathcal{L} = x_{max}$ with neural networks. To this end, we generate training and test data with a forward Euler discretization of the system (4.2). The whole solver can be downloaded from *https://github.com/mroberto166/MultilevelMachineLearning*. In the following
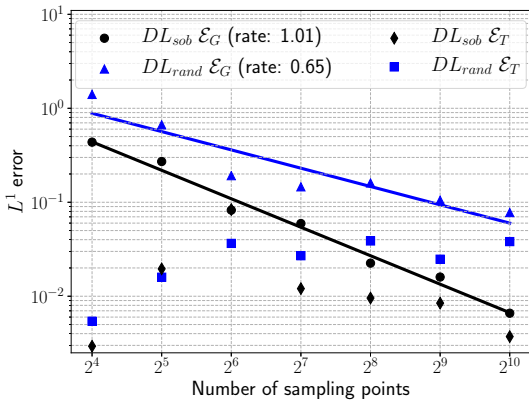

Figure 4: L^1 generalization and training errors for approximating the horizontal range x_{max} corresponding to the ODE system (4.2) using DL_{sob} as well as DL_{rand}. For the DL_{sob} the best result of the ensemble is reported while for the DL_{rand} the retrained average of the best ensemble result is shown.
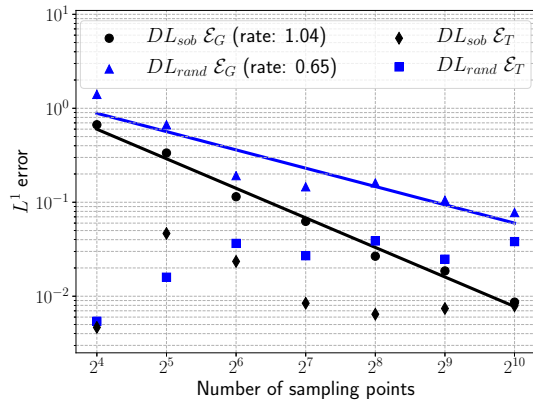
Figure 5: L^1 generalization and training errors for approximating the horizontal range x_{max} corresponding to the ODE system (4.2) using DL_{sob} as well as DL_{rand}. For both the DL_{sob} as well as the DL_{rand} the retrained average of the best ensemble result is reported.

experiment, the data sets are generated with a step size of $\Delta t = 0.00125$. In figure 4, we present the results for the training error and the generalization error with both the DL_{sob} as well as DL_{rand} algorithms, with varying size of the training set. From this figure, we see that the training errors are smaller for the DL_{sob} than for DL_{rand}. This is not explained by the bounds (3.16) and (3.9), as we do not estimate the training error in any particular way. On the other hand, the generalization gap (and the generalization error) for DL_{sob} is considerably smaller than the generalization error for DL_{rand} and decays at approximately the expected rate of 1, with respect to the number of training samples. This is in contrast to the generalization error for the DL_{rand} which decays with a rate of 0.65.

We remind the reader that the DL_{sob} errors are for the best performing network selected in the ensemble training, whereas the DL_{rand} errors are averages over the multiple retrainings. Could this difference in evaluating errors, which is completely justified by the theory presented here, play a role in how the errors behave? To test this, in figure 5, we compare the training and generalization errors for DL_{sob} and DL_{rand}, with both being averaged over 100 retrainings. There is a slight difference in the average training error for DL_{sob}, when compared to the ones shown in figure 4. In particular, the average training error does not decay as neatly as for a single network. On the other hand, the generalization error decays at rate 1 and is significantly smaller than the generalization error for DL_{rand}

4.4 Computational Finance: pricing for a European Call Option

In this experiment, we want to compute the "fair" price of a *European multi-asset basket call option*. We assume that there is a basket of underlying assets $S_1, \ldots, S_d$, which change in time according to the

14

multivariate geometric brownian motion (GBM), i.e.

$$dS_i = rS_i(t)dt + \sigma_i S_i(t)dW_i(t),$$
$$S_i(0) = (S_0)_i,$$

for all $i = 1, \ldots, d$, where r is called the risk-free interest rate, σ_i is the volatility and the Wiener processes W_i are correlated such that $d\langle W_i, W_j \rangle(t) = \rho_{ij}dt$ for all $i, j = 1, \ldots, d$. Additionally, the payoff function associated to our basket option is given as

$$\Lambda(\mathbf{S}) = \Lambda(S_1, \ldots, S_d).$$

The pricing function $V(t, \mathbf{S})$ is then the discounted conditional expectation of its payoff under an equivalent martingale measure $\tilde{P}$:

$$V(t, \mathbf{S}) = e^{-r(T-t)}\mathbb{E}_{\tilde{P}}[\Lambda(\mathbf{S}(T)) \mid \mathbf{S}(t) = \mathbf{S}].$$

However, using the Feynman-Kac Formula, one can show that the payoff satisfies a multidimensional Black-Scholes partial differential equation (BSPDE) ([4]) of the form:

$$\frac{\partial V}{\partial t} + \sum_{i=1}^{d} rS_i \frac{\partial V}{\partial S_i} + \frac{1}{2}\sum_{i=1}^{d} \sigma_i^2 S_i^2 \frac{\partial^2 V}{\partial S_i^2} + \sum_{i=1}^{d-1}\sum_{j=i+1}^{d} \rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 V}{\partial S_i \partial S_j} - rV = 0, \tag{4.4}$$

subject to the terminal condition $V(T, \mathbf{S}) = \Lambda(\mathbf{S})$.
Even though a general closed-form solution does not exist for the multi-dimensional BSPDEs, using a specific payoff function enables computing the solution in terms of an explicit formula. Therefore, we will consider the payoff function corresponding to the European geometric average basket call option:

$$\Lambda(\mathbf{S}(T)) = \max\{\left(\prod_{i=1}^{d} S_i(T)\right)^{1/d} - K, 0\}. \tag{4.5}$$

Using the fact that products of log-normal random variables are log-normal, one can derive the following for the initial value of the pricing function, i.e. for the discounted expectation at time $t = 0$:

$$V(0, \mathbf{S}) = e^{-rT}\mathbb{E}_{\tilde{P}}[\Lambda(\mathbf{S}(T)) \mid \mathbf{S}(0) = \mathbf{S}]. \tag{4.6}$$

This is of particular interest, as it denotes the value of the option at the time when it is bought. This provides the fair price for the European geometric average basket Call option in the Black-Scholes model. Using the particular payoff function (4.5), the closed-form solution to the multidimensional BSPDE (4.4) at point $(0, \mathbf{S})$ is then given as (Theorem 2 in [22]):

$$V(0, \mathbf{S}) = e^{-rT}(\tilde{s}e^{\tilde{m}}\Phi(d_1) - K\Phi(d_2)), \tag{4.7}$$

where Φ is the standard normal cumulative distribution function. Additionally

$$\nu = \frac{1}{d}\sqrt{\sum_{j=1}^{d}\left(\sum_{i=1}^{d}\sigma_{ij}^2\right)^2}, \qquad m = rT - \frac{1}{2d}\sum_{i=1}^{d}\sum_{j=1}^{d}\sigma_{ij}^2 T, \qquad \tilde{m} = m + \frac{1}{2}\nu^2,$$

$$\tilde{s} = \left(\prod_{i=1}^{d} S_i\right)^{1/d}, \qquad d_1 = \frac{\log(\frac{\tilde{s}}{K}) + m + \nu^2}{\nu}, \qquad d_2 = d_1 - \nu,$$

where the entries σ_{ij} of $\sigma \in \mathbb{R}^{d \times d}$ are the covariances (and variances) of the stock returns. For the subsequent experiments, we fix the following parameters: $\sigma = 10^{-5}\mathbb{1}$, $T = 5$, $K = 0.08$ and $r = 0.05$, where $\mathbb{1} \in \mathbb{R}^{d \times d}$ is the identity matrix.

The underlying asset prices S_i, at time $t = 0$ are drawn from $[0, 1]^d$. The underlying map for this simulation is given by the fair price $\mathcal{L}(\mathbf{S}) = V(0, \mathbf{S})$ of the basket option at time $t = 0$. We generate the

training data for both the DL_{sob} and the DL_{rand} algorithms by computing $\mathcal{L}(\mathbf{S})$ for either Sobol points or random points in Y, i.e. by evaluating the explicit formula (4.7) for each point in the training set. We note that this formula is an explicit solution of the underlying Black-Scholes PDE (4.4). In case no formula is available, for instance with a different payoff function, either a finite difference or finite element method approximating the underlying PDE (in low dimensions) or a Multi-Carlo approximation of the integral (4.6) can be used to generate the training data.

In figure 6, we plot the generalization error for the DL_{sob} with respect to the number of training points, till 2^{10} training points, for three different values of the underlying dimension d (number of assets in the basket). For concreteness, we consider $d = 5, 7, 9$ and see that the rate of decay of the generalization error does depend on the underlying dimension and ranges from 0.9 (for $d = 5$) to approximately 0.5 for $d = 9$. This is completely consistent with the bound (3.9) on the generalization gap as the effect of the logarithmic correction is stronger for higher dimensions. In fact, the logarithmic term is expected to dominate till at least $N \approx 2^d$ training points. Once we consider even more training points, the logarithmic term is weaker and a higher rate of decay can be expected, as predicted by (3.9). To test this, we plot the generalization error till 2^{12} training points for the 9-dimensional case ($d = 9$ in (4.4)), in figure 7. For the sake of comparison, we also plot the generalization errors with the DL_{rand} algorithm. From figure 7, we observe that indeed, the rate of decay of generalization errors for the DL_{sob} algorithm does increase to approximately 0.7, when the number of training points is greater than 2^9. In this regime, the DL_{sob} algorithm significantly outperforms the DL_{rand} algorithm providing a factor of 5 speed up and a significantly better convergence rate, even for this moderately high-dimensional problem.
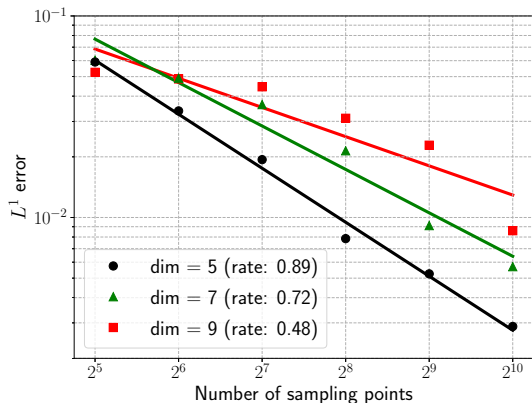

Figure 6: L^1 generalization errors for approximating the fair price (4.7) using DL_{sob} for different input dimensions of the underlying map.
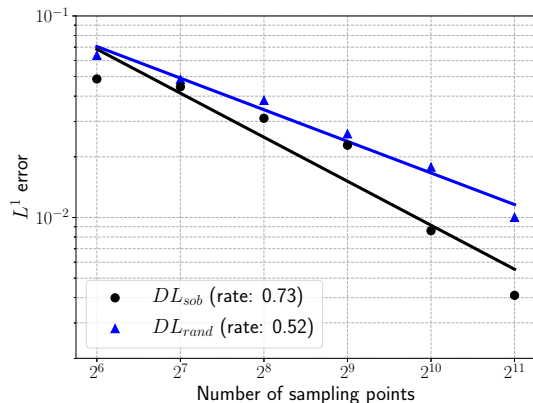
Figure 7: L^1 generalization error for approximating the fair price (4.7) for a fixed input dimensions of 9 using DL_{sob} compared to using DL_{rand}.

4.5 Computational Fluid Dynamics: flow past airfoils

In this section, we consider a prototypical example from computational fluid dynamics, namely that of a compressible flow past the well-studied RAE2822 airfoil [19]. The underlying parametric PDE (2.2) are the well-known compressible Euler equations of aerodynamics (Eqns (5.1) of [26]) in two space dimensions, which are considered in an exterior domain of a *perturbed* airfoil, parameterized by,

$$S(x; y) = \overline{S}(x) + \sum_{i=1}^{d} a_i \sin^4\left(\pi x^{\frac{\ln(0.5)}{\ln(x_{M_i})}}\right). \tag{4.8}$$

Here, $\overline{S} := [\overline{S}_L, \overline{S}_U]$ is the shape function for the reference RAE2822 airfoil (see figure 8(left)) and the airfoil is deformed (perturbed) by the well-known *Hicks-Henne bump functions* [27], further parameterized by a_i, x_{M_i} representing the amplitude and the location (maximum) of the perturbed basis functions, respectively.

The flow is defined by the following free-stream boundary conditions,

$$T^\infty = 1, \qquad M^\infty = 0.729, \qquad p^\infty = 1, \qquad \alpha = 2.31°,$$

where α denotes the angle of attack and T, p, M are the temperature, pressure and Mach-number of the incident flow.

The goal of the computation is to approximate the lift and drag coefficients of the flow:

$$C_L(y) = \frac{1}{K^\infty(y)} \int_S p(y)n(y) \cdot \hat{y}ds, \tag{4.9}$$

$$C_D(y) = \frac{1}{K^\infty(y)} \int_S p(y)n(y) \cdot \hat{x}ds, \tag{4.10}$$

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

We remark that this configuration is a prototypical configuration for aerodynamic shape optimization. Here, the underlying maps $\mathcal{L}$ (2.1) are the lift coefficient C_L and the drag coefficient C_D, realized as functions of a *design space* $Y = [0,1]^{2d}$. A variant of this problem was considered in recent papers [26, 25], where the underlying problem was that of uncertainty quantification (UQ) whereas we consider a shape optimization problem here.

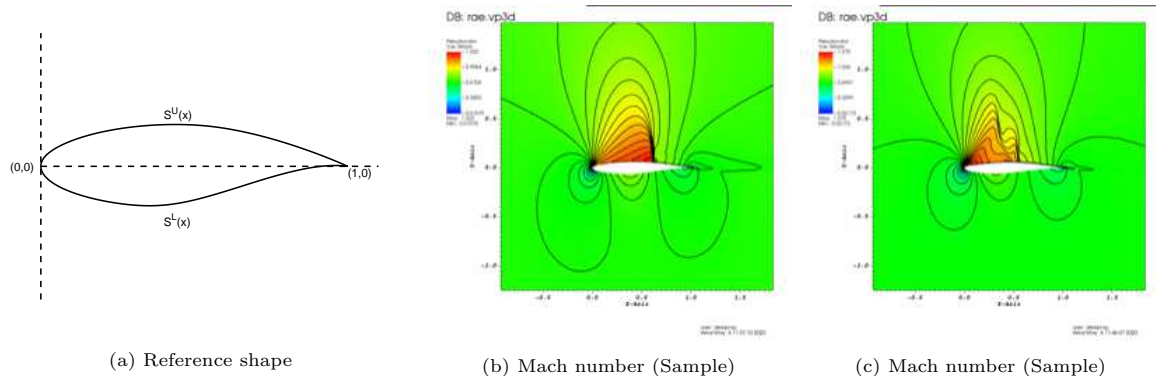(a) Reference shape   (b) Mach number (Sample)   (c) Mach number (Sample)

Figure 8: Flow past a RAE2822 airfoil with different airfoil shapes Left: Reference airfoil Center and Right: Flow visualized with Mach number for two different samples.

The training data is generated by solving the compressible Euler equations in the above domain, with the heavily used NEWTUN code. Two realizations of the flow (with two different airfoil shapes) are depicted in figure 8 (center and right).

We study the DL_{sob} and DL_{rand} algorithms for different sizes of the training set. In figure 9, we show the generalization errors for the drag coefficient with DL_{sob} algorithms for different numbers of shape variables (different dimensions $2d$) with $2d = 6$ and $2d = 10$. As seen from this figure, the generalization error for the 6-dimensional problem decays at a rate of 0.6. This is significantly smaller than the rate of 1, predicted from (3.9). However, it is unclear if the Hardy-Krause variation V_{HK} of the drag and the lift will be finite as the flow is very complicated and can even have shocks in this transsonic regime (see figure 8). So, it is very possible that the Hardy-Krause variation blows up in this case (see the discussion on this issue in [26]) and this will certainly reduce the rate of convergence in (3.9). Nevertheless, the DL_{sob} algorithm clearly outperforms the DL_{rand} algorithm for this problem. As expected the DL_{rand} converges at the rate of 0.5. Hence, at the considered resolutions, the DL_{sob} algorithm provides almost an order of magnitude speedup.

In figure 9, we also show the generalization errors for the drag coefficient with DL_{sob} algorithm when $2d = 10$, i.e for the 10-dimensional problem. In this case, the generalization error decays at a slightly worse rate of ≈ 0.56. This is not unexpected, given the logarithmic corrections in (3.9). Very similar results were seen for the lift coefficient and we omit them here.

Finally, we push the DL_{sob} algorithm to the limits by setting $2d = 20$, i.e. considering the 20-dimensional design space. From (3.9), we can see that even if the Hardy-Krause variation of the drag and

lift coefficents was finite, the logarithmic terms can dominate for such high dimensions till the unreasonable number of 2^{20} training points is reached. Will the DL_{sob} algorithm still outperform the DL_{rand} algorithm in this extreme situation? The answer is provided in figure 10, where we plot the generalization errors for the lift coefficient, with respect to number of training samples, for both the DL_{rand} and DL_{sob} algorithms. We observe from this figure that although both deep learning algorithms approximate this underlying map with errors, decaying at the rate of 0.5, the amplitude of the error is significantly smaller for the DL_{sob} algorithm than for the DL_{rand} algorithm, leading to approximate speedup of a factor of 5 with the former over the latter. Thus, even in this realistic problem with maps of very low regularity and with a very high underlying dimension, the deep learning algorithm based on low-discrepancy sequences as training points, performs much better than the standard deep learning algorithm, based on random training data.
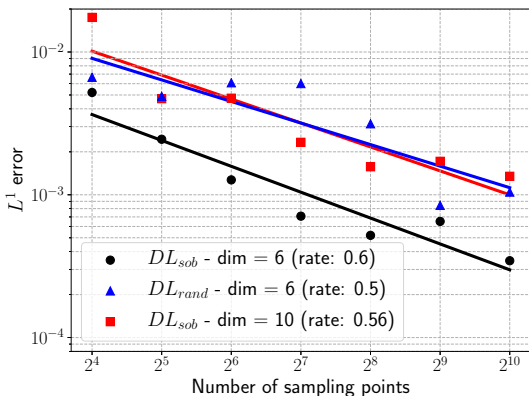

Figure 9: L^1 generalization errors for approximating the drag coefficient (4.10) using DL_{sob} for input dimensions of 6 and 10 compared to using DL_{rand} for a fixed input dimension of 6.
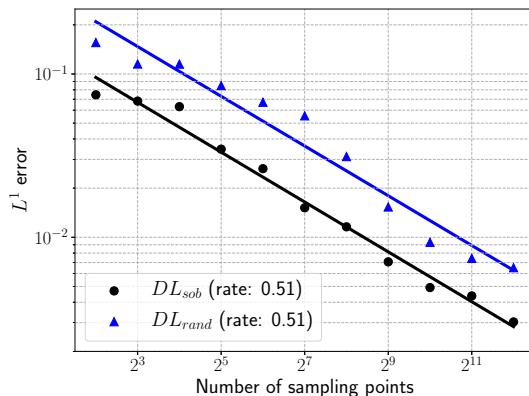
Figure 10: L^1 generalization errors for approximating the lift coefficient (4.9) for a fixed input dimensions of 20 using DL_{sob} compared to using DL_{rand}.

5 Conclusion

A key goal in scientific computing is that of simulation or prediction, i.e. evaluation of maps $\mathcal{L}$ (2.1) for different inputs. Often, evaluation of these maps require expensive numerical solutions of underlying ordinary or partial differential equations. In the last couple of years, deep learning algorithms have come to the fore for designing surrogates for these computationally expensive maps. However, the standard paradigm of supervised learning, based on randomly chosen training data, suffers from a serious bottleneck for these problems. This is indicated by the bound (3.16) on the resulting generalization error, i.e. large number of training samples are needed to obtain accurate surrogates. This makes the training process computationally expensive, as each evaluation of the map on a training sample might entail an expensive PDE solve.

In this article, we follow the recent paper [26] and propose an alternative deep learning algorithm 2.2 by considering low-discrepancy sequences (see definition 2.1) as the *training set*. We rigorously analyze this algorithm and obtain the upper bounds (3.7), (3.9) on the underlying generalization error (3.5) in terms of the training error (3.6) and a generalization gap, that for sufficiently regular functions (ones with bounded Hardy-Krause variation) and with sufficiently smooth activation functions for the neural network (2.9) is shown to decay at a linear rate (with dimension-dependent logarithmic corrections) with respect to the number of training samples. When compared to the standard deep learning algorithm with randomly chosen points, these bounds suggest that at least for maps with desired regularity, up to moderately high underlying dimensions, the proposed deep learning algorithm will significantly outperform the standard version.

We test these theoretically derived assertions in a set of numerical experiments that considered approximating maps arising in mechanics (ODEs), computational finance (linear PDEs) and computational

fluid dynamics (nonlinear PDEs). From the experiments, we observe that the proposed deep learning algorithm 2.2, with Sobol training points, is indeed accurate and significantly outperforms the one with random training points. The expected linear decay of the error is observed for problems with moderate dimensions. Even for problems with moderately high (up to 20) dimensions, the proposed algorithm led to much smaller amplitude of errors, than the one based on randomly chosen points. This gain is also seen for maps with low regularity, for which the upper bound (3.9) does not necessarily hold. Thus, the proposed deep learning algorithm is a very simple yet efficient alternative for building surrogates in the context of scientific computing.

The proposed deep learning algorithm can be readily extended to a *multi-level* version to further reduce the generalization error. This was already proposed in a recent paper [25], although no rigorous theoretical analysis was carried out. This multi-level version would be very efficient for approximating maps with low regularity. We also aim to employ the proposed deep learning algorithms in other contexts in computational PDEs and finance in future papers.

A key limitation of the theoretical analysis of the proposed algorithm 2.2, as expressed by the bound (3.9) on the generalization error, is in the very nature of this bound. The bound is on the so-called generalization gap and does not aim to estimate the training error (3.6) in any way. This is standard practice in theoretical machine learning [1], as the training error entails estimating the result of a stochastic gradient descent algorithm for a highly non-convex, very high-dimensional optimization problem and rigorous guarantees on this method are not yet available. Moreover, in practical terms, this is not an issue. Any reasonable machine learning procedure involves a systematic search of the hyperparameter space and finding suitable hyperparameter configurations that lead to the smallest training error. In particular, this training error is computed a posteriori. Thus, our bounds (3.9) imply that *as long as we train well, we generalize well.*

Another limitation of the proposed algorithm 2.2 is the logarithmic dependence on dimension in the bound (3.9) on the generalization gap. This dependence is also seen in practice (as shown in figures 6 and 9) and will impede the efficiency of the proposed algorithm for approximating maps in very high dimensions (say $d > 20$). However, this issue has been investigated in some depth in recent literature on Quasi-Monte Carlo methods [7] and references therein and a class of *high-order quasi Monte Carlo* methods have been proposed which can handle this *curse of dimensionality* at least for maps with sufficient regularity. Such methods could alleviate this issue in the current context and we explore them in a forthcoming paper.

Acknowledgements.

References

[1] S. Arora, R. Ge, B. Neyshabur, and Y. Zhang. Stronger generalization bounds for deep nets via a compression approach. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 254–263, 2018.

[2] K. Basu and A. B. Owen. Transformations and hardy–krause variation. *SIAM Journal on Numerical Analysis*, 54(3):1946–1966, 2016.

[3] R. E. Caflisch. Monte carlo and quasi-monte carlo methods. *Acta Numerica*, 7:1–49, 1998.

[4] G. Campolieti and R. N. Makarov. *Financial mathematics: a comprehensive treatment.* CRC Press, 2016.

[5] F. Cucker and S. Smale. On the mathematical foundations of learning. *Bulletin of the American Mathematical Society*, 39(1):1–49, 2002.

[6] J. Dick, F. Y. Kuo, and I. H. Sloan. High-dimensional integration: the quasi-monte carlo way. *Acta Numerica*, 22:133–288, 2013.

[7] J. Dick, F. Y. Kuo, Q. T. Le Gia, and C. Schwab. Multilevel higher order qmc petrov–galerkin discretization for affine parametric operator equations. *SIAM Journal on Numerical Analysis*, 54(4): 2541–2568, 2016.

[8] W. E and B. Yu. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6(1):1–12, 2018.

[9] W. E, J. Han, and A. Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics*, 5(4):349–380, 2017.

[10] W. E, C. Ma, and L. Wu. A priori estimates of the population risk for two-layer neural networks. *Communications in Mathematical Sciences*, 17(5):1407–1425, 2019.

[11] R. Evans, J. Jumper, J. Kirkpatrick, L. Sifre, T. Green, C. Qin, A. Zidek, A. Nelson, A. Bridgland, H. Penedones, et al. De novo structure prediction with deep-learning based scoring. *Annual Review of Biochemistry*, 77(363-382):6, 2018.

[12] R. Ghanem, D. Higdon, and H. Owhadi. *Handbook of uncertainty quantification*, volume 6. Springer, 2017.

[13] M. B. Giles. Multilevel monte carlo path simulation. *Operations Research*, 56(3):607–617, 2008.

[14] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

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

[16] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2(1):84–90, 1960.

[17] J. Han, A. Jentzen, and W. E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510, 2018.

[18] S. Heinrich. Multilevel monte carlo methods. In *International Conference on Large-Scale Scientific Computing*, pages 58–67. Springer, 2001.

[19] C. Hirsch, D. Wunsch, J. Szumbarski, J. Pons-Prats, et al. Uncertainty management for robust industrial design in aeronautics. Notes on Numerical Fluid Mechanics and Multidisciplinary Design. Springer, 2019.

[20] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML*, volume 37, pages 448–456, 2015.

[21] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[22] R. Korn and S. Zeytun. Efficient basket monte carlo option pricing via a simple analytical approximation. *Journal of Computational and Applied Mathematics*, 243:48–59, 2013.

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

[24] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[25] K. O. Lye, S. Mishra, and R. Molinaro. A multi-level procedure for enhancing accuracy of machine learning algorithms. *arXiv preprint arXiv:1909.09448*, 2019.

[26] K. O. Lye, S. Mishra, and D. Ray. Deep learning observables in computational fluid dynamics. *Journal of Computational Physics*, page 109339, 2020.

[27] D. A. Masters, N. J. Taylor, T. Rendall, C. B. Allen, and D. J. Poole. Geometric comparison of aerofoil shape parameterization methods. *AIAA Journal*, pages 1575–1589, 2017.

[28] S. Mishra. A machine learning framework for data driven acceleration of computations of differential equations. *Mathematics in Engineering*, 1:118, 2019.

[29] B. Neyshabur, Z. Li, S. Bhojanapalli, Y. LeCun, and N. Srebro. The role of over-parametrization in generalization of neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[30] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*, volume 63. SIAM, 1992.

[31] A. B. Owen. Monte carlo variance of scrambled net quadrature. *SIAM Journal on Numerical Analysis*, 34(5):1884–1910, 1997.

[32] A. B. Owen. Multidimensional variation for quasi-monte carlo. In *Contemporary Multivariate Analysis And Design Of Experiments: In Celebration of Professor Kai-Tai Fang's 65th Birthday*, pages 49–74. World Scientific, 2005.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *Workshop Proceedings of Neural Information Processing Systems*, 2017.

[34] A. Quarteroni, A. Manzoni, and F. Negri. *Reduced basis methods for partial differential equations: an introduction*, volume 92. Springer, 2015.

[35] M. Raissi and G. E. Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

[36] 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 arXiv:1808.04327*, 2018.

[37] C. E. Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[38] D. Ray and J. S. Hesthaven. An artificial neural network as a troubled-cell indicator. *Journal of Computational Physics*, 367:166–191, 2018.

[39] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge University Press, 2014.

[40] I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4):784–802, 1967.

[41] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML*, volume 70, pages 3424–3433, 2017.