Available online at www.sciencedirect.com

ScienceDirect

Comput. Methods Appl. Mech. Engrg. 374 (2021) 113575

Computer methods in applied mechanics and engineering

www.elsevier.com/locate/cma

Iterative surrogate model optimization (ISMO): An active learning algorithm for PDE constrained optimization with deep neural networks

Kjetil O. Lye[a], Siddhartha Mishra[b,*], Deep Ray[c], Praveen Chandrashekar[d]

[a] *Mathematics and Cybernetics, SINTEF, Forskningsveien 1, 0373 Oslo, Norway*
[b] *Seminar for Applied Mathematics (SAM), D-Math, ETH Zürich, Rämistrasse 101, Zürich-8092, Switzerland*
[c] *Department of Aerospace and Mechanical Engineering, University of Southern California, Los Angeles, USA*
[d] *TIFR Centre for Applicable Mathematics, Bangalore 560065, India*

Received 15 August 2020; received in revised form 31 October 2020; accepted 9 November 2020
Available online 4 December 2020

Abstract

We present a novel *active learning* algorithm, termed as *iterative surrogate model optimization* (ISMO), for robust and efficient numerical approximation of PDE constrained optimization problems. This algorithm is based on deep neural networks and its key feature is the iterative selection of training data through a feedback loop between deep neural networks and any underlying standard optimization algorithm. Numerical examples for optimal control, parameter identification and shape optimization problems for PDEs are provided to demonstrate that ISMO significantly outperforms a standard deep neural network based surrogate optimization algorithm as well as standard optimization algorithms.

Keywords: CFD; Deep learning; Optimization; Neural networks

1. Introduction

A large number of problems of interest in engineering reduce to the following *optimization problem*:

$$\text{Find} \quad \bar{y} = \arg\min_{y \in Y} G\left(\mathcal{L}(y)\right), \tag{1.1}$$

Here, $Y \subset \mathbb{R}^d$, the underlying map (*observable*) $\mathcal{L} : Y \to Z$, with either $Z \subset \mathbb{R}^m$ or is a subset of an infinite-dimensional Banach space and $G : Z \to \mathbb{R}$ is the *cost* or *objective function*. We can think of Y as a design or control space, with any $y \in Y$ being a vector of design or control parameters, that steers the system to minimize the objective function.

Many engineering systems of interest are modeled by partial differential equations (PDEs). Thus, evaluating the desired observable $\mathcal{L}$ and consequently, the objective function in (1.1), requires (approximate) solutions of the

* Corresponding author.
E-mail addresses: kjetil.olsen.lye@sintef.no (K.O. Lye), siddhartha.mishra@sam.math.ethz.ch (S. Mishra), deepray@usc.edu (D. Ray), praveen@tifrbng.res.in (P. Chandrashekar).

https://doi.org/10.1016/j.cma.2020.113575

underlying PDE. The resulting problem is termed as a *PDE constrained optimization* problem, [1,2] and references therein.

A representative example for PDE constrained optimization is provided by the problem of designing the wing (or airfoils) of an aircraft in order to minimize drag [3,4]. Here, the design variables are the parameters that describe the wing shape. The underlying PDEs are the compressible Euler or Navier–Stokes equations of gas dynamics and the objective function is the drag coefficient (while keeping the lift coefficient within a range) or the lift to drag ratio. Several other examples of PDE constrained optimization problems arise in fluid dynamics, solid mechanics and the geosciences [1].

A wide variety of numerical methods have been devised to solve optimization problems in general, and PDE constrained optimization problems in particular [1,2]. A large class of methods is iterative and requires evaluation of the gradients of the objective function in (1.1), with respect to the design (control) parameters y. These include first-order methods based on *gradient descent* and second-order methods such as quasi-newton and truncated Newton methods [5]. Evaluating gradients in the context of PDE constrained optimization often involves the (numerical) solution of *adjoints* (duals) of the underlying PDE [1]. Other types of optimization methods such as particle swarm optimization [6] and genetic algorithms [7] are *gradient free* and are particularly suited for optimization, constrained by PDEs with rough solutions.

Despite the well documented success of some of the aforementioned algorithms in the context of PDE constrained optimization, the solution of realistic optimization problems involving a large number of design parameters ($d \gg 1$ in (1.1)) and with the underlying PDEs set in complex geometries in several space dimensions or describing multiscale, multiphysics phenomena, remains a considerable challenge. This is mostly on account of the fact that applying the above methods entails evaluating the objective function and hence the underlying map $\mathcal{L}$ and its gradients a large number (in the range of 10^2–10^4) of times. As a single call to the underlying PDE solver can be very expensive, computing the underlying PDEs (and their adjoints) multiple times could be prohibitively expensive. This expense makes *robust optimization* that involves computing sensitivities of the minima to inputs such as the starting values of the optimizer [8], infeasible in most problems of practical interest.

One possible framework for reducing the computational cost of PDE constrained optimization problems is to use *surrogates* [9]. This approach 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*. Then, a *surrogate model* is constructed by designing a surrogate map, $\hat{\mathcal{L}} : Y \to Z$ such that $\hat{\mathcal{L}}(y) \approx \mathcal{L}(y)$, for all $y \in \mathcal{S}$. Finally, one runs a standard optimization algorithm such as gradient descent or its variants, while evaluating the functions and gradients in (1.1), with respect to the surrogate map $\hat{\mathcal{L}}$. This surrogate model will be effective as long as $\mathcal{L} \approx \hat{\mathcal{L}}$ in a suitable sense, for all $y \in Y$ 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 [10] and Gaussian process regression [11].

A particularly attractive class of such surrogate models are *deep neural networks* (DNNs) [12], i.e. functions formed by multiple compositions of affine transformations and scalar non-linear activation functions. Deep neural networks have been extremely successful at diverse tasks in science and engineering [13] such as at image and text classification, computer vision, text and speech recognition, autonomous systems and robotics, game intelligence and even protein folding [14].

Deep neural networks are also being increasingly used in different contexts in scientific computing. A very incomplete list of the rapidly growing literature includes the use of deep neural networks to approximate solutions of PDEs by so-called physics informed neural networks (PINNs) [15–20] and references therein, solutions of high-dimensional PDEs, particularly in finance [21–23] and references therein, improving the efficiency of existing numerical methods for PDEs, for instance in [24–26] and references therein.

A different approach is taken in recent papers [27–29], where the authors presented supervised deep learning algorithms to efficiently approximate *observables* (quantities of interest) for solutions of PDEs and applied them to speedup existing (Quasi)-Monte Carlo algorithms for forward Uncertainty quantification (UQ). These papers demonstrated that deep neural networks can result in effective surrogates for observables in the context of PDEs. Given the preceding discussion, it is natural to take a step further and ask if the resulting surrogates can be used for PDE constrained optimization. This indeed constitutes the central premise of the current paper.

Our first aim in this article is to present an algorithm for PDE constrained optimization that is based on combining standard (gradient based) optimization algorithms and deep neural network surrogates for the underlying maps in (1.1). The resulting algorithm, which we term *DNNopt*, is described and examined, both theoretically (with suitable and very stringent hypotheses on the underlying problem) and in several numerical experiments. We find that although *DNNopt* can be a viable algorithm for PDE constrained optimization and converges to a (local) minimum of the underlying optimization problem (1.1), it might require a large number of training samples for the deep neural network, which impedes cost efficiency of the overall algorithm. Moreover, it was found that this algorithm is not robust enough and can lead to a high *variance* or sensitivity of the resulting (approximate) minima, with respect to starting values for the underlying optimization algorithm. A careful analysis reveals that a key cause for these issues lies in the fact that the training set for the deep neural network surrogate is fixed *a priori*, based on global approximation requirements. This training set may not necessarily represent the subset (in parameter space) of minimizers of the objective function in (1.1) well, if at all.

To this end, the second and *main aim* of this paper is to propose a novel *active learning* procedure to *iteratively* augment training sets for training a sequence of deep neural networks, each providing successively better approximations of the underlying minimizers of (1.1). The additions to the training sets in turn, are based on local minimizers of (1.1), identified by running standard optimization algorithms on the neural network surrogate at the previous step of iteration. This feedback between training neural networks for the observable $\mathcal{L}$ in (1.1) and adding local optimizers as training points leads to the proposed algorithm, that we term as *iterative surrogate model optimization (ISMO)*. Thus, ISMO can be thought of as an example of an *active learning algorithm* [30], where the learner (deep neural network) queries the teacher (standard optimization algorithm) to iteratively identify training data that provides a better approximation of local optima. We test ISMO in several numerical experiments and observe that it outperforms the DNNopt algorithm readily, both in terms of the computational cost as well as in terms of the robustness, i.e., we observe significantly less sensitivity of the computed minima, with respect to starting values for the ISMO algorithm than for the DNNopt algorithm. Moreover, we also find that ISMO is able to significantly outperform existing gradient based optimization algorithms. Thus, ISMO is shown to be a very effective framework for PDE constrained optimization.

The rest of this paper is organized as follows: in Section 2, we put together preliminary material for this article. The DNNopt and ISMO algorithms for PDE constrained optimization are presented in Sections 3 and 4 , respectively and numerical experiments are reported in Section 5.

2. Preliminaries

2.1. The underlying constrained optimization problem

The basis of our constrained optimization problem is the following time-dependent parametric PDE,

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

Here, $Y \subset \mathbb{R}^d$ is the underlying parameter space that describes the design (control) variables $y \in Y$. For simplicity of notation and exposition, we set $Y = [0,1]^d$ for the rest of this paper. The spatial domain is labeled as $y \to D(y) \subset \mathbb{R}^{d_s}$ and $\mathbf{U} : [0,T] \times D \times Y \to \mathbb{R}^n$ 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 is specific examples. Moreover, L_b is a generic operator for imposing boundary conditions.

For the parametrized PDE (2.1), we define a generic form of the *parameters to observable* map:

$$\mathcal{L} : Y \to \mathbb{R}^m, \; Y \ni y \mapsto \mathcal{L}(y) = \mathcal{L}(y, \mathbf{U}) \in \mathbb{R}^m, \tag{2.2}$$

with m being the output dimension, representing m quantities of interest.

Finally, we define the cost (objective or goal) function G as $G \in C^2(\mathbb{R}^m; \mathbb{R})$. A very relevant example for such a cost function is given by,

$$G(\mathcal{L}(y)) := |\mathcal{L}(y) - \bar{\mathcal{L}}|^p, \tag{2.3}$$

for some target $\bar{\mathcal{L}} \in \mathbb{R}^m$, with $1 \leqslant p < \infty$ and $|.|$ denoting a vector norm. We note that $p = 2$ is the most popular choice in this setting and we will consider this choice of p in the rest of the paper.

The resulting constrained optimization problem is given by

$$\text{Find} \quad \bar{y} = \arg\min_{y \in Y} G\left(\mathcal{L}(y)\right), \tag{2.4}$$

with the observable defined by (2.2) and C^2-objective function, for instance the one defined in (2.3).

We also denote,

$$\mathcal{G}(y) = G(\mathcal{L}(y)), \quad \forall y \in Y, \tag{2.5}$$

for notational convenience.

2.2. Standard optimization algorithms

For definiteness, we restrict ourselves to the class of *(quasi)-Newton* algorithms as our standard optimization algorithm in this paper. We assume that the cost function $\mathcal{G}$ (2.5) is such that $\mathcal{G} \in C^1(Y)$. We note that this assumption implicitly requires some smoothness on the solutions of the parametric PDE (2.1) in order to obtain differentiability of the map $y \mapsto \mathcal{G}(y)$. A generic quasi-Newton algorithm for computing (approximate, local) minimizers of the optimization problem (2.4) has the following form,

Algorithm 2.1 (*Quasi-Newton Approximation of* (2.4))**.**

Inputs: Underlying cost function $\mathcal{G}$ (2.5), starting value $y_0 \in Y$ and starting Hessian $B_0 \in \mathbb{R}^{d\times d}$, tolerance parameter ε

Goal: Find (local) minimizer for the optimization problem (2.4).

At Step k: Given y_k, B_k

- Find search direction $p_k \in \mathbb{R}^d$ by solving

$$B_k p_k = -\nabla_y \mathcal{G}(y_k) \tag{2.6}$$

- Perform a *line search* (one-dimensional optimization):

$$\beta_k = \arg\min_{\beta\in[0,1]} \mathcal{G}(y_k + \beta p_k) \tag{2.7}$$

- Set $y_{k+1} = y_k + \beta_k p_k$
- Update the approximate Hessian,

$$B_{k+1} = \mathcal{F}\left(B_k, y_k, y_{k+1}, \nabla_y \mathcal{G}(y_k), \nabla_y \mathcal{G}(y_{k+1})\right) \tag{2.8}$$

- If $|y_{k+1} - y_k| < \varepsilon$, terminate the iteration, else continue.

Different formulas for the update of the Hessian (2.8) lead to different versions of the quasi-Newton Algorithm 2.1. For instance, the widely used BFGS algorithm [5] uses an update formula (2.8) such that the inversion step (2.6) can be directly computed with the well-known Sherman–Morrison formula, from a recurrence relation. Other versions of quasi-Newton algorithms, in general truncated Newton algorithms, use iterative linear solvers for computing the solution to (2.6). Note that quasi-Newton methods do not explicitly require the Hessian of the objective function in (2.4), but compute (estimate) it from the gradients in (2.8). Thus, only information on the gradient of the cost $\mathcal{G}$ (2.5) is required. Moreover, we need that the sequence of approximate optimizers $y_k \in Y$. This requires bound preserving versions of quasi-Newton algorithms, which are readily available [5].

We observe that implementing Algorithm 2.1 in the specific context of the optimization problem (2.4), constrained by the PDE (2.1), requires evaluating the map $\mathcal{L}$ and its gradients multiple times during each iteration. Thus, having a large number of iterations in Algorithm 2.1 entails a very high computational cost on account of the large number of calls to the underlying PDE solver.

2.3. Neural network surrogates

As mentioned in the introduction, we will employ neural network based surrogates for the underlying observable $\mathcal{L}$ (2.2) in order to reduce the computational cost of employing standard optimization Algorithm 2.1 for approximating the solutions of the optimization problem (2.4). To this end, we need the following ingredients.

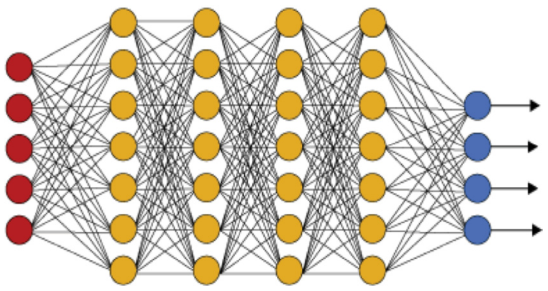


Fig. 1. An illustration of a (fully connected) deep neural network. The red neurons represent the inputs to the network and the blue neurons denote the output layer. They are connected by hidden layers with yellow neurons. Each hidden unit (neuron) is connected by affine linear maps between units in different layers and then with nonlinear (scalar) activation functions within units. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2.3.1. Training set

As is customary in supervised learning ([12] 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 in machine learning that the points in the training set $\mathcal{S}$ are chosen randomly from the parameter space Y, independently and identically distributed with the Lebesgue measure. However, we can follow recent papers [27,29] to choose *low discrepancy sequences* [31,32] such as Sobol or Halton sequences as training points in order to obtain better rates of convergence of the resulting generalization error. For Y in low dimensions, one can even choose (composite) Gauss quadrature points as viable training sets [19].

2.3.2. Deep neural networks

Given an input vector $y \in Y$, a feedforward neural network (also termed as a multi-layer perceptron), shown in Fig. 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 [12], 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 [12]. Popular choices for the activation function σ in (2.9) include the sigmoid function, the tanh function and the *ReLU* function defined by,

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

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 Fig. 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 kth 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

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

2.4. 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

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

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 (2.14) is minimized.

It is common in machine learning [12] 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 $q = 2$, with θ_W defined in (2.12). 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 [33].

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 $\mathcal{L}$ (2.2).
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 either randomly chosen or a low-discrepancy sequence or any other set of quadrature points in Y. 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 $\mathcal{L}$.

Remark 2.3. The cost of the deep learning Algorithm 2.2 includes the cost of generating the training samples $\mathcal{L}(y)$ for all $y \in \mathcal{S}$ and the cost of the training process, in the form of the stochastic gradient descent algorithm. Typically, for learning observables, the cost of generating the training data can be very high as expensive PDE solves have to be performed for each sample. On the other hand, given that the neural networks are of relatively small size, the training costs are significantly lower [27]. Thus, generating the training data is the dominant contributor to the cost budget for Algorithm 2.2. ■

3. Constrained optimization with DNN surrogates

Next, we combine the deep neural network surrogate $\mathcal{L}^*$, generated with the deep learning Algorithm 2.2, with the standard optimization Algorithm 2.1 in order to obtain the following algorithm,

Algorithm 3.1 (*DNNopt: A Deep Learning Based Algorithm for PDE Constrained Optimization*).

Inputs Parametrized PDE (2.1), observable (2.2), cost function (2.5), training set $\mathcal{S} \subset Y$ (either randomly chosen or suitable quadrature points such as low discrepancy sequences), standard optimization Algorithm 2.1.

Goal Compute (approximate) minimizers for the PDE constrained optimization problem (2.4).

Step 1 Given training set $\mathcal{S} = \{y_i\}$, with $y_i \in Y$ for $1 \leqslant i \leqslant N$, generate the deep neural network surrogate map $\mathcal{L}^*$ by running the deep learning Algorithm 2.2. Set $\mathcal{G}^*(y) = G(\mathcal{L}^*(y))$ for all $y \in Y$, with the cost function G defined in (2.5).

Step 2 Draw M random starting points $\tilde{y}_1, \dots, \tilde{y}_M$ in Y. For each $1 \leqslant j \leqslant M$, run the standard optimization Algorithm 2.1 with cost function $\mathcal{G}^*$ and starting values $\tilde{y}_j$ till tolerance to obtain a set $\bar{y}_j$ of approximate minimizers to the optimization problem (2.4).

The output of the DNNopt Algorithm 3.1 is the set $\bar{y}_j$, for $1 \leqslant j \leqslant M$, of approximate minimizers of the approximate cost function $\mathcal{G}^*$. One can further prune this set of minimizers by choosing those $\Delta M < M$ $\bar{y}_j$'s with the ΔM smallest values of $\mathcal{G}^*(\bar{y}_j)$. We denote this set as $\bar{y}_k$ with $1 \leqslant k \leqslant \Delta M$. As an additional step, it might be necessary to evaluate $\mathcal{G}(\bar{y}_k)$, by running an expensive PDE solve, for these ΔM optimizers. This can help in verification of the approximate optimizers being true optimizers and also in quantifying the sensitivity of the optimizers to starting values.

The following remarks about Algorithm 3.1 are in order,

Remark 3.2. The quasi-Newton optimization Algorithm 2.1 requires derivatives of the cost function with respect to the input parameters y. This boils down to evaluation of $\nabla_y \mathcal{L}^*$. As $\mathcal{L}^*$ is a neural network of form (2.9), these derivatives can be readily and very cheaply computed with backpropagation. ■

Remark 3.3. The overall cost of DNNopt Algorithm 3.1 is expected to be dominated by the cost of generating the training data $\mathcal{L}(y_i)$, for $y_i \in \mathcal{S}$, as this involves calls to the underlying and possibly expensive PDE solver. In practice, for observables $\mathcal{L}$, the cost of training the network can be much smaller than the cost of generating the training data, whereas the cost of evaluating the map $\mathcal{L}^*$ and its gradient $\nabla_y \mathcal{L}^*$ is negligible (see Table 5 of the recent paper [27] for a comparison of these costs for a realistic example in aerodynamics). Thus, the cost of running the optimization Algorithm 2.1 for the neural network surrogate is negligible, even for a large number of iterations. ■

Remark 3.4. In step 2 of the DNNopt algorithm, we require that M different random $\tilde{y}_i \in Y$ be used as starting values for the standard optimization Algorithm 2.1. The simplest choice is to set $M = 1$. However, as stated in the introduction, we are interested in *robust optimization*. This requires that the sensitivity of optimizers with respect to the inputs to an optimization algorithm be small. A key input to the optimization Algorithm 2.1 is the starting value. Given that most PDE constrained optimization problems of practical interest involve *non-convex* cost functions, an optimization algorithm such as Algorithm 2.1 can only approximate local minima. Hence, different starting values might lead to convergence to different local minima, with very different optimal cost functions. Thus, computing sensitivity with respect to starting values is of great importance for non-convex optimization.

However, computing measures of sensitivity like the variance or range of the optimal costs requires multiple starting values for the standard optimization Algorithm 2.1. In general, this is not feasible as each iteration for the optimization algorithm, for each different value, requires expensive PDE solves. The overall cost is prohibitively expensive.

In this context, the fact that DNNopt Algorithm 3.1 only requires evaluations of the neural network $\mathcal{L}^*$ and its gradients, these evaluations are extremely cheap (see [27]) and allow us to run possibly a large number of starting values for the optimization algorithm. This allows us to calculate sensitivity of the optimizers to starting values and represents a key advantage of the DNNopt algorithm over standard optimization algorithms such as Algorithm 2.1. ■

3.1. Analysis of DNNopt Algorithm 3.1

A priori, DNNopt Algorithm 3.1 appears very attractive for robust PDE constrained optimization, when compared to the standard optimization Algorithm 2.1. However, a careful analysis already exposes the following issues with this algorithm.

As discussed in Remark 3.3, the cost of DNNopt Algorithm 3.1 is dominated by the cost of generating the training data, i.e., evaluating $\mathcal{L}(y_i)$ for $1 \leqslant y_i \leqslant N$. Thus, the DNNopt will only be efficient, when compared to the

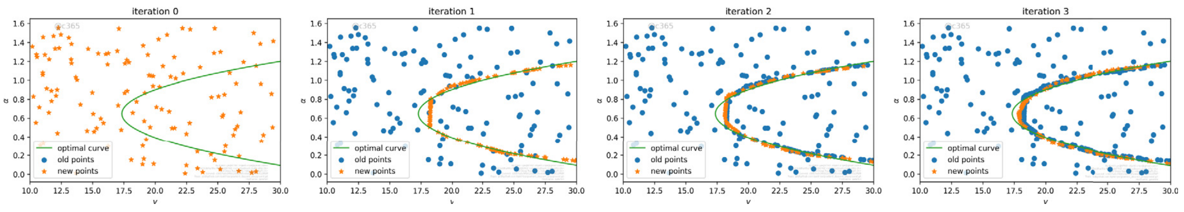


Fig. 2. Evolution of training sets $\mathcal{S}_k$, for different iterations k, for the ISMO Algorithm 4.1 for the optimal control of projectile motion example. The newly added training points $\bar{\mathcal{S}}_k$ are the orange dots, while existing training points $\mathcal{S}_k$ are blue dots. The exact minimizers are placed on the green parabola. Note how most of the newly added training points lie on this parabola.

standard optimization Algorithm 2.1, if the number N of training samples can be kept reasonably low for a given accuracy for the optimizers. This might not always be the case as the neural network is trained to minimize the mismatch between the underlying map $\mathcal{L}$ and the surrogate $\mathcal{L}^*$ in an integral (or possibly pointwise sense). On the other hand, minimizers and minimum values of the cost function are based on derivatives of the underlying map $\mathcal{L}$. In principle, one can require that the training of neural network in Algorithm 2.2 be performed with respect to a loss function (2.15) that also involves mismatch in the derivatives between the map $\mathcal{L}$ and surrogate $\mathcal{L}^*$. This in turn requires us to generate training data $\nabla_y \mathcal{L}$, evaluated at training points $y_i \in Y$. Computing such gradients necessitates the use of adjoint methods [3], which in practice are complicated to implement and very expensive.

Another potential problem for DNNopt Algorithm 3.1 is due to the fact that the training set $\mathcal{S}$ is chosen either at random or as equi-distributed low-discrepancy sequences and does not contain any information about the minimizers, which might lie on very localized sub-manifolds of the parametric domain Y. This is indeed the case in most problems of interest and can be viewed very clearly from Fig. 2(Left) for a model problem. Thus, it is unclear if the DNNopt algorithm would outperform a standard optimization algorithm.

Given the above discussion, the efficiency of the DNNopt algorithm hinges on the following question. How many training samples N are needed in order to achieve a desired accuracy for the optimization procedure ? This question is very hard to answer in full generality as it entails estimating the errors of approximating minima of a non-convex function. Most theoretical results on optimization require some convexity hypothesis. Moreover, analyzing Algorithm 3.1 entails a subtle interaction between non-convex optimization and bounds on generalization error for deep neural networks. Both tasks are not possible with currently available theoretical machinery.

Nevertheless, it is possible to identify suitable hypothesis on the cost function and neural network approximation such that we can prove some rigorous bounds on the accuracy of Algorithm 3.1. We present such hypothesis below with the caveat that they are very stringent and will not hold in problems of practical interest that we investigate in the numerical experiments later in the paper.

3.1.1. *Error bounds for Algorithm 3.1 in a special case*

For the rest of this paper, we set $|.| = |.|_\infty$ which is the max vector norm. We start with suitable hypothesis on the underlying map $\mathcal{L}$ in (2.4).

$$(H1) \quad \mathcal{L} : Y \to Z \subset \mathbb{R}, \quad \mathcal{L} \in W^{2,\infty}(Y), \quad \mathcal{C}_{\mathcal{L}} := \|\mathcal{L}\|_{W^{2,\infty}(Y)}. \tag{3.1}$$

Next, we have the following hypothesis on the function G that appears in the cost function (2.5),

$$(H2) \quad G : Z \to X \subset \mathbb{R}, \quad G \in W^{2,\infty}(Z), \quad \mathcal{C}_G := \|G\|_{W^{2,\infty}(Z)}. \tag{3.2}$$

Recalling that the cost function in (2.5) is given by $\mathcal{G} = G(\mathcal{L})$, the hypotheses $H1$ and $H2$ imply that $\mathcal{G} \in W^{2,\infty}(Y; X)$ with $\mathcal{C}_{\mathcal{G}} = \|\mathcal{G}\|_{W^{2,\infty}(Y)}$ such that $\mathcal{C}_{\mathcal{G}} \leqslant \mathcal{C}_G \mathcal{C}_{\mathcal{L}}$. However, we need further hypothesis on this cost function given by,

$$\begin{aligned} &(H3) \quad \mathcal{G} \text{ has exactly one global minimum at } \bar{y} \text{ and no other local minima,} \\ &(H4)\ \nabla \mathcal{G} : Y \to \mathcal{Y} \subset \mathbb{R}^d \text{ is invertible with inverse } \nabla \mathcal{G}^{-1} : \mathcal{Y} \to Y \text{ and } \nabla \mathcal{G}^{-1} \in W^{1,\infty}(\mathcal{Y}), \\ &\quad \mathcal{C}_{-1} := \|\nabla \mathcal{G}^{-1}\|_{W^{1,\infty}} \end{aligned} \tag{3.3}$$

Note that the assumptions $H3$ and $H4$ are automatically satisfied by strictly convex functions. Thus, these hypotheses can be considered as a slight generalization of strict convexity.

We are interested in analyzing DNNopt Algorithm 3.1, with the underlying map and cost function satisfying the above hypothesis. Our aim is to derive estimates on the distance between the global minimizer $\bar{y}$ of the cost function $\mathcal{G}$ (see hypothesis $H3$) and the approximate minimizers $\bar{y}_i$, for $1 \leqslant i \leqslant N$, generated by DNNopt Algorithm 3.1. However, the standard training procedure for neural networks (see Algorithm 2.2) consists of minimizing loss functions in L^p-norms and this may not suffice in controlling pointwise errors near extremas of the underlying function. Thus, we need to train the neural network to minimize stronger norms that can control the derivative.

To this end, we fix $N \in \mathbb{N}$ and choose the training set $\mathcal{S} = \{y_i\}$ for $1 \leqslant i \leqslant N$ such that each $y_i \in Y$ and the training points satisfy,

$$(H5) \quad Y \subset \cup_{i=1}^{N} B\left(y_i, N^{-\frac{1}{d}}\right), \tag{3.4}$$

with $B(y, r)$ denoting the $|.|_\infty$ ball, centered at y, with diameter r.

The simplest way to enforce hypothesis $H5$ is to divide the domain Y into N cubes, centered at y_i, with diameter $N^{-\frac{1}{d}}$.

On this training set, we assume that for any parameter $\theta \in \Theta$, the neural network $\mathcal{L}_\theta \in W^{2,\infty}(Y)$ by requiring that the activation function $\sigma \in W^{2,\infty}(\mathbb{R})$ and consider the following (*Lipschitz*) loss function:

$$\mathcal{E}_{N,T}^{Lip}(\theta) = \max_{1 \leqslant i \leqslant N} \left(|\mathcal{L}(y_i) - \mathcal{L}_\theta(y_i)| + |\nabla\mathcal{L}(y_i) - \nabla\mathcal{L}_\theta(y_i)|\right) \tag{3.5}$$

Then, a stochastic gradient descent algorithm is run in order to find

$$\theta^* = \operatorname*{argmin}_{\theta \in \Theta} \mathcal{E}_{N,T}^{Lip}(\theta). \tag{3.6}$$

Note that the gradients of the Lipschitz loss function (3.5) can be computed by backpropagation. We denote the trained neural network as $\mathcal{L}^* = \mathcal{L}_{\theta^*}$ and set

$$\mathcal{E}_{N,T}^{Lip,*} = \mathcal{E}_{N,T}^{Lip}(\theta^*) \tag{3.7}$$

Finally, we need the following assumption on the standard optimization Algorithm 2.1 i.e., for all starting values $y_i \in \mathcal{S}$, optimization Algorithm 2.1 converges (with a finite number of iterations) to a local minimum $\bar{y}_i \in Y$ of the cost function $\mathcal{G}^* = \mathcal{G}(\mathcal{L}^*)$ such that

$$(H6) \quad \nabla\mathcal{G}^*(\bar{y}_i) \equiv 0. \tag{3.8}$$

Note that we are not assuming that $\mathcal{G}^*$ is convex, hence, only convergence to a local minimum can be realized.

We have the following bound on the minimizers, generated by DNNopt Algorithm 3.1 in this framework,

Lemma 3.5. *Let the underlying map $\mathcal{L}$ in (2.4) satisfy $H1$, the function G satisfy $H2$, the cost function $\mathcal{G}$ satisfy $H3$ and $H4$. Let $\mathcal{L}^*$ be a neural network surrogate for $\mathcal{L}$, generated by Algorithm 2.2, with training set $\mathcal{S} = \{y_i\}$, for $1 \leqslant i \leqslant N$, such that the training points satisfy $H5$ and the neural network is obtained by minimizing the Lipschitz loss function (3.5). Furthermore, for random starting values $\tilde{y}_j \in Y$, with $1 \leqslant j \leqslant M$ as inputs to the standard optimization Algorithm 2.1, let $\bar{y}_j$ denote the local minimizers of $\mathcal{G}^* = G(\mathcal{L}^*)$, generated by the Algorithm 2.1 and satisfy $H6$, then we have the following bound,*

$$|\bar{y} - \bar{y}_j| \leqslant (1 + \mathcal{C}_\mathcal{L})\mathcal{C}_{-1}\mathcal{C}_G\left(\mathcal{E}_{N,T}^{Lip,*} + (\mathcal{C}_\mathcal{L} + \mathcal{C}_{\mathcal{L}^*})N^{-\frac{1}{d}}\right), \quad \forall j, \tag{3.9}$$

with constants $\mathcal{C}_{\mathcal{L},-1,G}$ are defined in (3.1)–(3.3), $\mathcal{C}_{\mathcal{L}^*} = \|\mathcal{L}^*\|_{W^{2,\infty}}$ and $\mathcal{E}_{N,T}^{Lip,*}$ is defined through (3.5), (3.7).

Proof. Fix any $1 \leqslant j \leqslant N$ and let $\bar{y}_j$ be the minimizer of $\mathcal{G}^*$, generated by the standard optimization Algorithm 2.1 for starting value $\tilde{y}_j$. We have the following inequalities,

$$\begin{aligned}
|\bar{y} - \bar{y}_j| &= \left|\nabla\mathcal{G}^{-1}(\nabla\mathcal{G}(\bar{y})) - \nabla\mathcal{G}^{-1}\left(\nabla\mathcal{G}(\bar{y}_j)\right)\right| \quad \text{by } (H4), \\
&\leqslant \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}) - \nabla\mathcal{G}(\bar{y}_j)\right|, \quad \text{by } (H4), \\
&= \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}_j)\right|, \quad \text{by } (H3), \\
&= \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}_j) - \nabla\mathcal{G}^*(\bar{y}_j)\right|, \quad \text{by } (H6),
\end{aligned}$$

Using the straightforward calculation $\nabla \mathcal{G}(y) = G'(\mathcal{L}(y))\nabla\mathcal{L}(y)$ and similarly for $\nabla\mathcal{G}^*$, in the above inequality yields,

$$\begin{aligned}|\bar{y} - \bar{y}_j| &\leqslant \mathcal{C}_{-1}\left|G'(\mathcal{L}(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}^*(\bar{y}_j)\right| \\ &\leqslant \underbrace{\mathcal{C}_{-1}\left|G'(\mathcal{L}(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j)\right|}_{T_1} \\ &+ \underbrace{\mathcal{C}_{-1}\left|G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}^*(\bar{y}_j)\right|}_{T_2}\end{aligned} \tag{3.10}$$

We estimate the terms $T_{1,2}$ as follows,

$$\begin{aligned}T_1 &= \mathcal{C}_{-1}\left|G'(\mathcal{L}(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j)\right| \\ &\leqslant \mathcal{C}_{-1}\mathcal{C}_{\mathcal{L}}\mathcal{C}_G|\mathcal{L}(\bar{y}_j) - \mathcal{L}^*(\bar{y}_j)|, \quad \text{by } H1, H2,\end{aligned} \tag{3.11}$$

and

$$\begin{aligned}T_2 &= \mathcal{C}_{-1}\left|G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}^*(\bar{y}_j))\nabla\mathcal{L}^*(\bar{y}_j)\right| \\ &\leqslant \mathcal{C}_{-1}\mathcal{C}_G|\nabla\mathcal{L}(\bar{y}_j) - \nabla\mathcal{L}^*(\bar{y}_j)|, \quad \text{by } H2,\end{aligned} \tag{3.12}$$

As $\bar{y}_j \in Y$, by the hypothesis $H5$ on the training set, there exists an $1 \leqslant i_j \leqslant N$ such that $y_{i_j} \in \mathcal{S}$ and $|\bar{y}_j - y_{i_j}| \leqslant N^{-\frac{1}{d}}$. Hence, we can further estimate $T_{1,2}$ from the following inequalities,

$$\begin{aligned}|\mathcal{L}(\bar{y}_j) - \mathcal{L}^*(\bar{y}_j)| &\leqslant |\mathcal{L}(\bar{y}_j) - \mathcal{L}(y_{i_j})| + |\mathcal{L}^*(\bar{y}_j) - \mathcal{L}^*(y_{i_j})| + |\mathcal{L}(y_{i_j}) - \mathcal{L}^*(y_{i_j})| \\ &\leqslant \mathcal{E}_{N,T}^{Lip,*} + (\mathcal{C}_{\mathcal{L}} + \mathcal{C}_{\mathcal{L}^*})\, N^{-\frac{1}{d}} \quad \text{by } H5 \text{ and (3.5), (3.7),}\end{aligned} \tag{3.13}$$

Here, we recall that $\mathcal{C}_{\mathcal{L}^*} = \|\mathcal{L}^*\|_{W^{2,\infty}}$. Similarly, we have,

$$\begin{aligned}|\nabla\mathcal{L}(\bar{y}_j) - \nabla\mathcal{L}^*(\bar{y}_j)| &\leqslant |\nabla\mathcal{L}(\bar{y}_j) - \nabla\mathcal{L}(y_{i_j})| + |\nabla\mathcal{L}^*(\bar{y}_j) - \nabla\mathcal{L}^*(y_{i_j})| + |\nabla\mathcal{L}(y_{i_j}) - \nabla\mathcal{L}^*(y_{i_j})| \\ &\leqslant \mathcal{E}_{N,T}^{Lip,*} + (\mathcal{C}_{\mathcal{L}} + \mathcal{C}_{\mathcal{L}^*})\, N^{-\frac{1}{d}} \quad \text{by } H5 \text{ and (3.5), (3.7),}\end{aligned} \tag{3.14}$$

Applying (3.13) and (3.14) into (3.11) and (3.12) and then substituting the result into (2.14) yield the desired inequality (3.9) □

As stated before, we compute M approximate optimizers by DNNopt Algorithm 3.1 in order to compute sensitivity of the optimization with respect to starting values. To this end, we can measure sensitivity by either computing the *range* of minimizers i.e.,

$$\text{range}(\mathcal{G}) := \max_{1\leqslant i,j\leqslant M} |\mathcal{G}(\bar{y}_i) - \mathcal{G}(\bar{y}_j)|, \tag{3.15}$$

or an empirical approximation to the standard deviation,

$$\text{std}(\mathcal{G}) := \sqrt{\frac{1}{M}\sum_{i=1}^{M}|\mathcal{G}(\bar{y}_i) - \bar{\mathcal{G}}|^2}, \quad \bar{\mathcal{G}} := \frac{1}{M}\sum_{i=1}^{M}\mathcal{G}(\bar{y}_i). \tag{3.16}$$

A trivial application of the estimate (3.9), together with (3.1), (3.2) yields,

$$\max\{\text{range}(\mathcal{G}), \text{std}(\mathcal{G})\} \leqslant 2(\mathcal{C}_{\mathcal{L}} + \mathcal{C}_{\mathcal{L}}^2)\mathcal{C}_{-1}\mathcal{C}_G^2\left(\mathcal{E}_{N,T}^{Lip,*} + (\mathcal{C}_{\mathcal{L}} + \mathcal{C}_{\mathcal{L}^*})\, N^{-\frac{1}{d}}\right) \tag{3.17}$$

In order to simplify notation while tracking constants, by rescaling the underlying functions, we can assume that $\mathcal{C}_{\mathcal{L}}, \mathcal{C}_G \leqslant \frac{1}{3}$, then the bound (3.9) simplifies to

$$|\bar{y} - \bar{y}_j| \leqslant \frac{4\mathcal{C}_{-1}}{9}\left(\mathcal{E}_{N,T}^{Lip,*} + \left(\frac{1}{3} + \mathcal{C}_{\mathcal{L}^*}\right)N^{-\frac{1}{d}}\right) \tag{3.18}$$

Next, we define a *well-trained network* as a deep neural network $\mathcal{L}^*$, generated by the deep learning Algorithm 2.2, which further satisfies for any training set $\mathcal{S} = \{y_i\}$, with $1 \leqslant i \leqslant N$, the assumptions,

$$\mathcal{C}_{\mathcal{L}^*} = \|\mathcal{L}^*\|_{W^{2,\infty}} \leqslant \frac{1}{3}, \qquad \mathcal{E}_{N,T}^{Lip,*} \leqslant \frac{1}{3}N^{-\frac{1}{d}}. \tag{3.19}$$

Hence, for a well-trained neural network, the bound (3.18) further simplifies to,

$$|\bar{y} - \bar{y}_j| \leqslant \frac{4\mathcal{C}_{-1}}{9} N^{-\frac{1}{d}} \tag{3.20}$$

In particular, the bound (3.20) ensures convergence of all the approximate minimizers $\bar{y}_i$, generated by DNNopt Algorithm 3.1, to the underlying minimizer, $\bar{y}$, of the optimization problem (2.4) as the number of training samples $N \to \infty$.

3.1.2. Discussion on the assumptions in Lemma 3.5

The estimates (3.9), (3.20) need several assumptions on the underlying map, cost function and trained neural network to hold. We have the following comments about these assumptions,

- The assumptions $H1$ and $H2$ entail that the underlying map $\mathcal{L}$ and function G are sufficiently regular i.e., $W^{2,\infty}$. A large number of underlying maps for PDE constrained optimization, particularly for elliptic and parabolic PDEs satisfy these assumptions [1].
- The assumptions $H3$ and $H4$ essentially amount to requiring that the cost function $\mathcal{G}$ is strictly convex. It is essential here to mention that such convexity hypothesis do not hold for most problems of practical interest as the observable $\mathcal{L}$ need not be convex. Nevertheless, it is standard in optimization theory [5] to prove guarantees on the algorithms for convex cost functions, while expecting that similar estimates will hold for approximating *local minima* of non-convex functions. This is especially true for very high-dimensional highly non-convex optimization problems in machine learning. See for instance the celebrated paper [33], in which the authors present the hugely successful ADAM gradient descent algorithm and only prove convergence guarantees in the convex case, even though ADAM is mostly employed for non-convex optimization problems.
- The hypothesis $H5$ on the training set is an assumption on how the training set *covers* the underlying domain [34]. It can certainly be relaxed from a deterministic to a random selection of points with some further notational complexity, see [35] for space filling arguments. It also stems from an *equidistribution* requirement on the training set and can be satisfied by low-discrepancy sequences such as Sobol points.
- The use of Lipschitz loss functions (3.5) for training the neural network is necessitated by the need to control the neural network pointwise. We can also use a *Sobolev* loss function of the form,

$$\mathcal{E}_{N,T}^{Sob}(\theta) = \frac{1}{N}\left(\sum_{i=1}^{N} |\mathcal{L}(y_i) - \mathcal{L}_\theta(y_i)|^s + \sum_{i=1}^{N} |\nabla\mathcal{L}(y_i) - \nabla\mathcal{L}_\theta(y_i)|^s + \sum_{i=1}^{N} |\nabla^2\mathcal{L}(y_i) - \nabla^2\mathcal{L}_\theta(y_i)|^s\right), \tag{3.21}$$

  with $s > d$. By Morrey's inequality, minimizing the above loss function will automatically control the differences between the (derivatives of) the underlying map and the trained neural network, pointwise and an estimate, similar to (3.9), can be obtained.

  As mentioned before, using an L^p-loss function such as (2.15) may be insufficient in providing control on pointwise differences between minima of the underlying cost function and approximate minima generated by the DNNopt Algorithm 3.1. Nevertheless, we emphasize that in practice, implementing Lipschitz loss functions such as (3.5) or Sobolev losses such as (3.21) is not feasible in practice as it requires values of the gradient (and higher derivatives) of the observable $\mathcal{L}$ at training points. Evaluating such gradients (and higher derivatives) requires computing adjoint equations of (2.1), which are complicated and very expensive to solve.
- The hypothesis $H6$ on the standard optimization Algorithm 2.1 is only for notational convenience and can readily be relaxed to an approximation i.e., $|\nabla\mathcal{G}^*(\bar{y}_i)| \leqslant \varepsilon$, for each i and for a small tolerance $\varepsilon \ll 1$. This merely entails a minor correction to the estimate (3.9). Quasi-Newton algorithms are well known to converge to such approximate local minima [5].
- The point of assuming *well-trained networks* in the sense of (3.19) is to simplify the estimates (3.9), (3.17). The first inequality in (3.19) is an assumption on uniform bounds on the trained neural networks (and their gradients) with respect to the number of training samples. The constant $1/3$ is only for notational convenience and can be replaced by any other constant. We want to emphasize that this assumption is strong and may not always hold. On the other hand, it can be monitored in practice by computing the Hessian of the trained Neural network, which is readily done by backpropagation at the end of training. Moreover, the estimate on

the second derivative can be estimated by the weights of the trained neural network. If one wants to enforce this bound strictly, then the weights either need to be clipped or a bound preserving optimization algorithm such as BFGS can be used.
On the other hand, the second inequality in (3.19) is an assumption on the training error which greatly serves to simplify the complexity estimates but is not necessary. The essence of this assumption is to require that training errors are smaller than the generalization gap. In general, there are no rigorous estimates on the training error with a stochastic gradient method as a very high-dimensional non-convex optimization problem is being solved.

- In Lemma 3.5, we have assumed that the neural network $\mathcal{L}^* \in W^{2,\infty}(Y)$. This rules out the use of ReLU activation function but allows using other popular activation functions such as the hyperbolic tangent.

The preceding analysis shows that even in a very special case, with very stringent assumptions on the cost function and trained neural network, DNNopt Algorithm 3.1 suffers from an essential drawback. This can be seen from the right hand side of (3.9), (3.20), where the error with respect to approximating the minimum $\bar{y}$ of the underlying cost function in (2.4), decays with a dimension dependent exponent $\frac{1}{d}$, in the number of training samples (and hence complexity of the algorithm). This exponent implies a very strong *curse of dimensionality* for DNNopt Algorithm 3.1. In particular, for optimization problems in even moderate dimensions, there could be a slow decay of the error as well as a large variance, see estimate (3.17) on the range and standard deviation, of the computed minimizers with the DNNopt algorithm, making it very sensitive to the starting values and other hyperparameters of the underlying optimization Algorithm 2.1. This slow rate of decay necessitates a large number of training samples, significantly increasing the cost of the algorithm and reducing its competitiveness with respect to standard optimization algorithms. Given that the analysis in the last section requires very strong hypothesis on both the underlying problem as well on the neural network approximations, it is reasonable to expect that the same curse of dimensionality and slow convergence might occur also for most practical problems where these hypothesis are violated. This compels us to search for an alternative algorithm that can be more efficient in this context. We do so in the next section.

4. Iterative surrogate model optimization (ISMO)

Our starting point is the assertion that one of the issues with the DNNopt Algorithm 3.1 is the fact that the training set $\mathcal{S}$ for the deep learning Algorithm 2.2 is chosen *a priori* and does not reflect any information about the location of (local) minima of the underlying cost function in (2.4). It is very likely that incorporating such information will increase the accuracy of the deep learning algorithm around minima. We propose the following iterative algorithm to incorporate such information,

Algorithm 4.1 (*ISMO: Iterative Surrogate Model Optimization Algorithm for PDE Constrained Optimization*)**.**

Inputs Parametrized PDE (2.1), observable (2.2), cost function (2.5), *initial* training set $\mathcal{S}_0 \subset Y$ with $\mathcal{S}_0 = \{y_i^0\}$ for $1 \leqslant i \leqslant N_0$ (either randomly chosen or suitable quadrature points such as low discrepancy sequences), standard optimization Algorithm 2.1, hyperparameters N_0, ΔN with $\Delta N \leqslant N_0$

Goal Compute (approximate) minimizers for the PDE constrained optimization problem (2.4).

Step $k = 0$ Given initial training set $\mathcal{S}_0 = \{y_i^0\}$, with $y_i^0 \in Y$ for $1 \leqslant i \leqslant N_0$, generate the deep neural network surrogate map $\mathcal{L}_0^*$ by running the deep learning Algorithm 2.2. Set $\mathcal{G}_0^*(y) = G(\mathcal{L}_0^*(y))$ for all $y \in Y$, with the cost function G defined in (2.5).
Given *batch size* $\Delta N \leqslant N_0$, draw ΔN random starting values $\tilde{y}_1^0, \ldots, \tilde{y}_{\Delta N}^0$ in Y. Run standard optimization Algorithm 2.1 with $\tilde{y}_j^0$ as starting values and cost function $\mathcal{G}_0^*$ till tolerance, to obtain the set $\bar{\mathcal{S}}_0 = \{\bar{y}_j^0\}$, for $1 \leqslant j \leqslant \Delta N$, of approximate minimizers to the optimization problem (2.4). Set

$$\mathcal{S}_1 = \mathcal{S}_0 \cup \bar{\mathcal{S}}_0 \tag{4.1}$$

Step $k \geqslant 1$ Given training set $\mathcal{S}_k$, generate a new deep neural network surrogate map $\mathcal{L}_k^*$ by running the deep learning Algorithm 2.2. Set $\mathcal{G}_k^*(y) = G(\mathcal{L}_k^*(y))$ for all $y \in Y$, with the cost function G defined in (2.5).
Draw ΔN random starting values $\tilde{y}_1^k, \ldots, \tilde{y}_{\Delta N}^k$.

Run standard optimization Algorithm 2.1 with $\tilde{y}_j^k$, $1 \leqslant j \leqslant \Delta N$, as starting values and with cost function $\mathcal{G}_k^*$ till tolerance, to obtain the set $\bar{\mathcal{S}}_k = \{\bar{y}_j^k\}$, for $1 \leqslant j \leqslant \Delta N$, of approximate minimizers to the optimization problem (2.4). Set

$$\mathcal{S}_{k+1} = \mathcal{S}_k \cup \bar{\mathcal{S}}_k \tag{4.2}$$

- If for some tolerance $\varepsilon \ll 1$,

$$\left| \frac{1}{\Delta N} \sum_{j=1}^{\Delta N} \mathcal{G}(\bar{y}_j^k) - \frac{1}{\Delta N} \sum_{j=1}^{\Delta N} \mathcal{G}(\bar{y}_j^{k-1}) \right| \leqslant \varepsilon, \tag{4.3}$$

then terminate the iteration. Else, continue to step $k+1$.

Remark 4.2. The key difference between ISMO Algorithm 4.1 and DNNopt Algorithm 3.1 lies in the structure of the training set. In contrast to the DNNopt algorithm, where the training set was chosen a priori, the training set in the ISMO algorithm is augmented at every iteration, by adding points that are identified as approximate local minimizers (by the standard optimization Algorithm 2.1) of the cost function $\mathcal{G}_k^*$, corresponding to the surrogate neural network model $\mathcal{L}_k^*$. Thus, a sequence of independent neural network surrogates is generated, with each successive iteration likely producing a new DNN surrogate with greater accuracy around local minima of the underlying cost function. Given this feedback between neural network training and optimization of the underlying cost function, we consider ISMO as an example of an *active learning* algorithm [30], with the deep learning Algorithm 2.2, playing the role of the learner, querying standard optimization Algorithm 2.1, which serves as the *teacher* or *oracle*, for obtaining further training data. ■

4.1. *Analysis of the ISMO Algorithm 4.1*

Does the ISMO Algorithm 4.1 lead to more accuracy than the DNNopt Algorithm 3.1? We will investigate this issue in some generality in the numerical experiments presented later. Performing a rigorous theoretical analysis and comparison between ISMO and DNNopt in generality is not possible currently. On the other hand, in Section 3.1.1, we had considered a very special case with strong hypothesis on the cost function and neural network approximation and shown that the DNNopt algorithm suffers from a curse of dimensionality, see estimate (3.9). At least in this very special case, with strong assumptions that may not hold in practice, can we prove bounds on the accuracy of the ISMO Algorithm 4.1 and show that it outperforms the DNNopt algorithm. We do so below.

4.1.1. *Error bounds for ISMO Algorithm 4.1 in a special case*

We will investigate this question within the framework used for the analysis of the DNNopt algorithm in Section 3.1. We recall that a modified Lipschitz loss function (3.5) was used for training the underlying deep neural network in the supervised learning Algorithm 2.2. We will continue to use this Lipschitz loss function. Hence, at any step k in Algorithm 4.1, with training set $\mathcal{S}_k$, the following loss function is used,

$$\mathcal{E}_T^{Lip,k}(\theta) = \max_{y_i \in \mathcal{S}_k} |\mathcal{L}(y_i) - \mathcal{L}_\theta(y_i)| + |\nabla\mathcal{L}(y_i) - \nabla\mathcal{L}_\theta(y_i)| \tag{4.4}$$

Then, a stochastic gradient descent algorithm is run in order to find

$$\theta_k^* = \operatorname*{argmin}_{\theta \in \Theta} \mathcal{E}_T^{Lip,k}(\theta). \tag{4.5}$$

We denote the trained neural network as $\mathcal{L}_k^* = \mathcal{L}_{\theta_k^*}$ and set

$$\mathcal{E}_T^{Lip,k,*} = \mathcal{E}_T^{Lip}(\theta_k^*) \tag{4.6}$$

We have the following bound on the minimizers, computed with the ISMO Algorithm 4.1,

Lemma 4.3. *Let the underlying map $\mathcal{L}$ in (2.4) satisfy $H1$ with $\mathcal{C}_\mathcal{L} \leqslant \frac{1}{3}$, the function G satisfy $H2$ with constant $\mathcal{C}_G \leqslant \frac{1}{3}$, the cost function $\mathcal{G}$ satisfy $H3$ and $H4$. Let the neural networks at every step k in Algorithm 4.1 be trained with the Lipschtiz loss functions (4.4) and local minimizers of the cost function $\mathcal{G}_k^*$, computed with standard* *optimization Algorithm 2.1 satisfy $H6$. Let the trained neural network $\mathcal{L}_0^*$ be well-trained in the sense of (3.19) and the number of initial training points, $N_0 = \#(\mathcal{S}_0)$ be chosen such that*

$$N_0 \geqslant \left(\frac{\mathcal{C}}{\sigma_0}\right)^d, \quad \text{for some } \sigma_0 < 1, \tag{4.7}$$

with constant $\mathcal{C} = \frac{4\mathcal{C}_{-1}}{9}$.

Given σ_0 in (4.7), define σ_k for $k \geqslant 1$ by,

$$\sigma_k = \mathcal{E}_T^{Lip,k,*} + \left(\frac{1}{3} + \mathcal{C}_{\mathcal{L}_k^*}\right)\frac{\sigma_{k-1}}{(\Delta N)^{\frac{1}{d}}}. \tag{4.8}$$

Under the assumption that the local minimizers $\bar{y}_j^k$, for $1 \leqslant j \leqslant \Delta N$ and at every $k \geqslant 0$ also satisfy,

$$(H7) \quad \bar{y}_j^k \in \cup_{i=1}^{\Delta N} B\left(\bar{y}_i^{k-1}, \frac{\sigma_{k-1}}{(\Delta N)^{\frac{1}{d}}}\right), \tag{4.9}$$

then the approximate minimizers $\bar{y}_j^k$, for all $k \geqslant 1$, $1 \leqslant j \leqslant \Delta N$ satisfy the bound,

$$|\bar{y} - \bar{y}_j^k| \leqslant \mathcal{C}\sigma_k, \tag{4.10}$$

with σ_k defined in (4.8).

In particular, if at every step k, the trained neural network $\mathcal{L}_k^*$ is well-trained *i.e., it satisfies,*

$$\mathcal{C}_{\mathcal{L}^*} = \|\mathcal{L}_k^*\|_{W^{2,\infty}} \leqslant \frac{1}{3}, \qquad \mathcal{E}_{N,T}^{Lip,k,*} \leqslant \frac{1}{3}\frac{\sigma_{k-1}}{(\Delta N)^{\frac{1}{d}}}, \tag{4.11}$$

then, the minimizers, generated by ISMO Algorithm 4.1 satisfy the bound,

$$|\bar{y} - \bar{y}_j^k| \leqslant \frac{\mathcal{C}^k \sigma_0}{(\Delta N)^{\frac{k}{d}}}, \quad \forall k \geqslant 1. \tag{4.12}$$

Proof. We observe that step $k = 0$ of the ISMO Algorithm 4.1 is identical to the DNNopt Algorithm 3.1, with training set $\mathcal{S}_0$ and N_0 training samples. As we have required that all the hypothesis, $H1 - H6$ for Lemma 3.5 holds and the neural network $\mathcal{L}_0^*$ is well-trained in the sense of (3.19), we can directly apply estimate (3.20) to obtain that

$$|\bar{y} - \bar{y}_j^0| \leqslant \mathcal{C}N_0^{-\frac{1}{d}} \leqslant \sigma_0 < 1, \quad \text{by (4.7)}. \tag{4.13}$$

We will prove the bound (4.10) by induction. Assume that (4.10) holds for step $k-1$. Given the training set $\mathcal{S}_k$, we obtain the trained neural network $\mathcal{L}_k^*$ by minimizing the Lipschitz loss function (4.4) and with starting values $\bar{y}_j^{k-1}$, for $1 \leqslant j \leqslant \Delta N$, we run standard optimization Algorithm 2.1 to generate approximate minimizers $\bar{y}_j^k$. We proceed as in the proof of Lemma 3.5 to calculate the following,

$$\begin{aligned}
|\bar{y} - \bar{y}_j^k| &= \left|\nabla\mathcal{G}^{-1}\left(\nabla\mathcal{G}(\bar{y})\right) - \nabla\mathcal{G}^{-1}\left(\nabla\mathcal{G}(\bar{y}_j^k)\right)\right| \quad \text{by } (H4),\\
&\leqslant \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}) - \nabla\mathcal{G}(\bar{y}_j^k)\right|, \quad \text{by } (H4),\\
&= \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}_j^k)\right|, \quad \text{by } (H3),\\
&= \mathcal{C}_{-1}\left|\nabla\mathcal{G}(\bar{y}_j^k) - \nabla\mathcal{G}_k^*(\bar{y}_j^k)\right|, \quad \text{by } (H6),\\
&= \mathcal{C}_{-1}\left|G'(\mathcal{L}(\bar{y}_j^k))\nabla\mathcal{L}(\bar{y}_j^k) - G'(\mathcal{L}_k^*(\bar{y}_j^k))\nabla\mathcal{L}_k^*(\bar{y}_j^k)\right|\\
&\leqslant \underbrace{\mathcal{C}_{-1}\left|G'(\mathcal{L}(\bar{y}_j^k))\nabla\mathcal{L}(\bar{y}_j^k) - G'(\mathcal{L}_k^*(\bar{y}_j^k))\nabla\mathcal{L}(\bar{y}_j^k)\right|}_{T_3} + \underbrace{\mathcal{C}_{-1}\left|G'(\mathcal{L}_k^*(\bar{y}_j^k))\nabla\mathcal{L}(\bar{y}_j) - G'(\mathcal{L}_k^*(\bar{y}_j^k))\nabla\mathcal{L}_k^*(\bar{y}_j^k)\right|}_{T_4}
\end{aligned}$$

As in the proof of Lemma 3.5, we can estimate,

$$T_3 \leqslant \mathcal{C}_{-1}\mathcal{C}_{\mathcal{L}}\mathcal{C}_G|\mathcal{L}(\bar{y}_j^k) - \mathcal{L}_k^*(\bar{y}_j^k)| \leqslant \frac{\mathcal{C}_{-1}}{9}|\mathcal{L}(\bar{y}_j^k) - \mathcal{L}_k^*(\bar{y}_j^k)|, \quad \text{by } H1, H2,$$

$$T_4 \leqslant \mathcal{C}_{-1}\mathcal{C}_G|\nabla\mathcal{L}(\bar{y}_j^k) - \nabla\mathcal{L}_k^*(\bar{y}_j^k)| \leqslant \frac{\mathcal{C}_{-1}}{3}|\mathcal{L}(\bar{y}_j^k) - \mathcal{L}_k^*(\bar{y}_j^k)|, \quad \text{by } H2.$$

By the hypothesis $H7$ on the approximate minimizers, there exists an $1 \leqslant i_{j,k} \leqslant \Delta N$ such that $\bar{y}^{k-1}_{i_{j,k}} \in \mathcal{S}_k$ and $|\bar{y}^k_j - \bar{y}^{k-1}_{i_{j,k}}| \leqslant \frac{\sigma_{k-1}}{(\Delta N)^{\frac{1}{d}}}$. Hence, we can further estimate from the Lipschitz loss function (4.4), completely analogously to the estimates (3.13) and (3.14) to obtain that,

$$|\bar{y} - \bar{y}^k_j| \leqslant \mathcal{C}\left(\mathcal{E}^{Lip,k,*}_T + \left(\frac{1}{3} + \mathcal{C}_{\mathcal{L}^*_k}\right)\frac{\sigma_{k-1}}{(\Delta N)^{\frac{1}{d}}}\right),$$

which with definition (4.8) is the desired inequality (4.10) for step k.

A straightforward application of the definitions (4.11) for well-trained networks $\mathcal{L}^*_k$ for $k \geqslant 1$, together with a recursion on the inequality (4.10) yields the bound (4.12). □

Remark 4.4. In addition to the assumptions $H1$–$H6$ in Section 3.1, we also need another assumption $H7$ for obtaining bounds on the minimizers, generated by the ISMO algorithm. This assumption amounts to requiring that during the iterations, the optimizers of (4.4), at a given step k, remain close to optimizers of (4.4), at the previous step $k-1$. This seems to be a *continuity* requirement on the minimizers. It implies as long as the first step $k = 0$ is able to provide a good approximation to the underlying (global) minimum and successive iterations improve the approximation of this global minimum. On the other hand, for non-convex cost functions, it could happen that such an assumption is violated near local minima. However, given the fact that the starting values for the standard optimization algorithm, at any iteration step $k \geqslant 1$ of ISMO Algorithm 4.1, are chosen randomly, we would expect that this provides a pathway for the algorithm to escape from unfavorable local minima. ■

Remark 4.5. The assumptions on *well-trained networks* (4.11), at each step of ISMO Algorithm 4.1 are for conceptual and notational simplicity. An estimate, analogous to (4.12), can be proved without resorting to assuming (4.11). ■

The estimates (3.20) and (4.12) provide a quantitative comparison of accuracy between DNNopt Algorithm 3.1 and *active learning* ISMO Algorithm 4.1, in this very special case, covered by Lemmas 3.5 and 4.3. To realize this, we further assume that the ISMO algorithm converges to the desired tolerance within K iterations and that the *batch size* $\Delta N = cN_0$, for some fraction $c \leqslant 1$. Hence, the accuracy of the ISMO algorithm is quantified by (4.12) as,

$$|\bar{y} - \bar{y}^K_j| \leqslant \frac{\mathcal{C}^K \sigma_0}{c^{\frac{K}{d}} N_0^{\frac{K}{d}}}. \tag{4.14}$$

As we have assumed that the cost of both DNNopt and ISMO algorithms is dominated by the cost of generating the training samples, the cost of the ISMO algorithm with K iterations is determined by the total number of training samples i.e., $(1 + cK)N_0$. For exactly the same number of training samples, the accuracy of the DNNopt Algorithm 3.1 is given by the estimate (3.20) as,

$$|\bar{y} - \bar{y}_j| \leqslant \frac{\mathcal{C}}{(1 + cK)^{\frac{1}{d}} N_0^{\frac{1}{d}}}. \tag{4.15}$$

As (fixed) $N_0 \gg 1$ and $c \approx 1$, as long as $\mathcal{C} \sim \mathcal{O}(1)$, we see that the estimate (4.14) suggests an *exponential decay* of the approximation error of the underlying minimum w.r.t. the size of the training set for the ISMO algorithm, in contrast to the *algebraic decay* for the DNNopt algorithm, at-least within the framework of Lemmas 3.5 and 4.3.

This contrast can also be directly translated to the *range* (3.15) and *standard deviation* (3.16) of the approximate minimizers for both algorithms. Straightforward calculations with the definitions (3.15), (3.16) and estimates (4.14), (4.15) reveal an exponential decay for the ISMO algorithm of the form

$$\max\{\mathrm{range}(\mathcal{G}), \mathrm{std}(\mathcal{G})\} \sim \mathcal{O}\left(N_0^{-\frac{K}{d}}\right), \tag{4.16}$$

whereas there is an algebraic decay for the DNNopt algorithm of the form,

$$\max\{\mathrm{range}(\mathcal{G}), \mathrm{std}(\mathcal{G})\} \sim \mathcal{O}\left((KN_0)^{-\frac{1}{d}}\right). \tag{4.17}$$

Table 1
Training parameters for the experiments done in Section 5.

Width	Depth	Regularizer	Optimizer	Loss function	Learning rate	Activation function
20	8	L^2: $7.8 \cdot 10^{-7}$	Adam	MSE	0.001	ReLU

These estimates (4.16) and (4.17) suggest that the ISMO algorithm may have significantly lower variance (sensitivity) of approximate minimizers of the underlying optimization problem (2.4), when compared to the DNNopt algorithm. In particular, these estimates highlight the role of the iterated feedback between the learner (the deep learning Algorithm 2.2) and the oracle (the standard optimization Algorithm 2.1) in providing the training data, that exponentially improves the approximation of the underlying minima, in each successive step. However, it is essential to point out that these theoretical results were derived in a very special case with stringent hypotheses, which may not hold in practice. Hence, the comparison between DNNopt and ISMO, in full generality, can only be performed with a suite of numerical experiments that we present below.

5. Numerical experiments

5.1. Details of implementation

The implementation of both DNNopt Algorithm 3.1 and ISMO Algorithm 4.1, is performed in Python utilizing the Tensorflow framework [36] with Keras [37] for the machine learning specific parts, while SciPy [38] is used for handling optimization of the objective function. The code is open-source, and can be downloaded from https://github.com/kjetil-lye/iterative_surrogate_optimization. New problem statements can be readily implemented without intimate knowledge of the algorithm.

The architecture and hyperparameters for the underlying neural networks in both algorithms are specified in Table 1. To demonstrate the robustness of the proposed algorithms, we train an ensemble of neural networks with 10 different starting values and evaluate the resulting statistics.

At each iteration k of ISMO Algorithm 4.1, we have to train the Neural network $\mathcal{L}_k^*$ with ADAM algorithm. As is well known, a stochastic gradient algorithm such as ADAM can be very oscillatory near local minima and deciding when to terminate the training is important. To this end, we use *early stopping* i.e. the loss function (2.15) is monitored at every iteration of ADAM and the training is stopped if the loss function has not decreased in the last $\hat{K}$ iterations (epochs). We use $\hat{K} = 50$ in all our simulations below.

By utilizing the job chaining capability of the IBM Spectrum LSF system [39], we are able to create a process-based system where each iteration of the ISMO algorithm is run as a separate job. The novel use of job chaining enables us to differentiate the job parameters (number of compute nodes, runtime, memory, etc.) for each step of the algorithm, without wasting computational resources.

5.2. Optimal control of projectile motion

This problem was proposed in a recent paper [28] as a prototype for using deep neural networks to learn observables for differential equations and dynamical systems. The motion of a projectile, subjected to gravity and air drag, is described by the following system of ODEs,

$$\frac{d}{dt}\mathbf{x}(t;y) = \mathbf{v}(t;y), \qquad \mathbf{x}(0;y) = \mathbf{x}_0(y), \tag{5.1}$$

$$\frac{d}{dt}\mathbf{v}(t;y) = -F_D(\mathbf{v}(t;y);y)\frac{\mathbf{v}}{|\mathbf{v}|_2} - g\mathbf{e}_2, \qquad \mathbf{v}(0;y) = \mathbf{v}_0(y), \tag{5.2}$$

where $F_D = \frac{1}{2m}\rho C_d \pi r^2 \|v\|^2$ denotes the drag force, the initial velocity is set to $\mathbf{v}(0;y) = [v_0\cos(\alpha), v_0\sin(\alpha)]$ and the various parameters are listed in Table 2.

The observable $\mathcal{L}$ of interest is the *horizontal range* $x_{\max}$ of the simulation:

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

Table 2
Parameters for the projectile motion experiment in Section 5.

$\mathbf{x}_0$	g	C_D	m	r	ρ	Δt
(0.2, 0.5)	9.81	0.1	0.142	0.22	1.1455	0.01

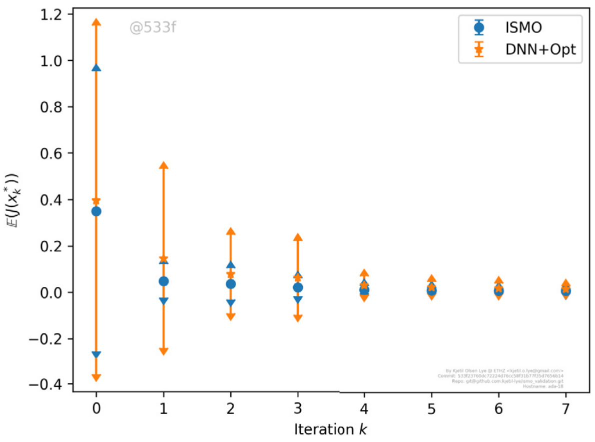


Fig. 3. The objective function vs. number of iterations of the DNNopt and ISMO algorithms for the optimal control of projectile motion. The mean of the objective function and mean ± standard deviation (3.16) are shown.

Here, $y = [\alpha, v_0]$ denotes the vector of *control parameters*. The aim of this *model problem* is to find initial release angle $\alpha \in [0, \pi/2]$ and initial velocity $v_0 \in [10, 30]$ such that the horizontal range x_{max} is exactly $x_{ref} = 15$. Thus, we have an optimal control problem, constrained by the ODE (5.1), with control parameters $y \in [0, 1]^2$. The underlying objective function is given by,

$$\mathcal{G}(y) = \frac{1}{2}|x_{max}(y) - x_{ref}|^2. \tag{5.3}$$

The advantage of this model problem is that the optimal control parameters, i.e., minimizers of (5.3), can be easily computed through cheap numerical experiments, which corresponds to points, lying on a parabola-like curve in the Y-space, shown in Fig. 2 (Left). Moreover, one can cheaply generate training data by solving the ODE system (5.1) with a standard forward Euler time discretization.

We run the DNNopt Algorithm 3.1 on this problem with hyperparameters listed in Table 1 and plot the mean $\bar{\mathcal{G}}$ (3.16) of the cost function (5.3), as well as the mean ± standard deviation (3.16) (as a measure of the robustness/sensitivity) of the algorithm in Fig. 3. For this figure, at iteration k, the size of the training set $\mathcal{S}$ in Algorithm 3.1 is given by $N_k = N_0 + k\Delta N$, with $N_0 = 32$ and $\Delta N = 16$. Moreover, we set $M_k = N_k$ in Algorithm 3.1 in order to compute sensitivity statistics. From Fig. 3, we see that the mean of the cost function, computed by the DNNopt algorithm converges to the underlying minimum $\mathcal{G} = 0$, with increasing number of training samples. However, the standard deviation (3.16) decays at a slow rate. In particular, the standard deviation is still high, for instance when $k = 3$ or $k = 4$. This high variance implies that the optimizers, computed by the DNNopt Algorithm 3.1 may not be robust. Note that the mean–standard deviation for the objective function, might be negative on account of non-zero standard deviation near the minimum (zero).

We also run ISMO Algorithm 4.1 for this problem, with initial $N_0 = 32$ training samples and a batch size of $\Delta N = 16$ in Algorithm 4.1. The resulting mean (and mean ± standard deviation) of the cost function is plotted in Fig. 3. From this figure, we see that the mean of the cost function quickly converges to 0. Moreover, and in contrast to the DNNopt algorithm, the standard deviation with ISMO, converges very rapidly to zero. This example also brings out a key advantage with ISMO, namely although the mean of the objective function is only slightly lower with ISMO than DNNopt in this example, the standard deviation is much lower. Hence, ISMO, at the same cost, is significantly more robust, than the DNNopt algorithm.

Why is ISMO more robust than DNNopt in this case? As explained before, we believe that the reason lies in the iterative feedback between the learner (supervised learning Algorithm 2.2) that queries the teacher (standard

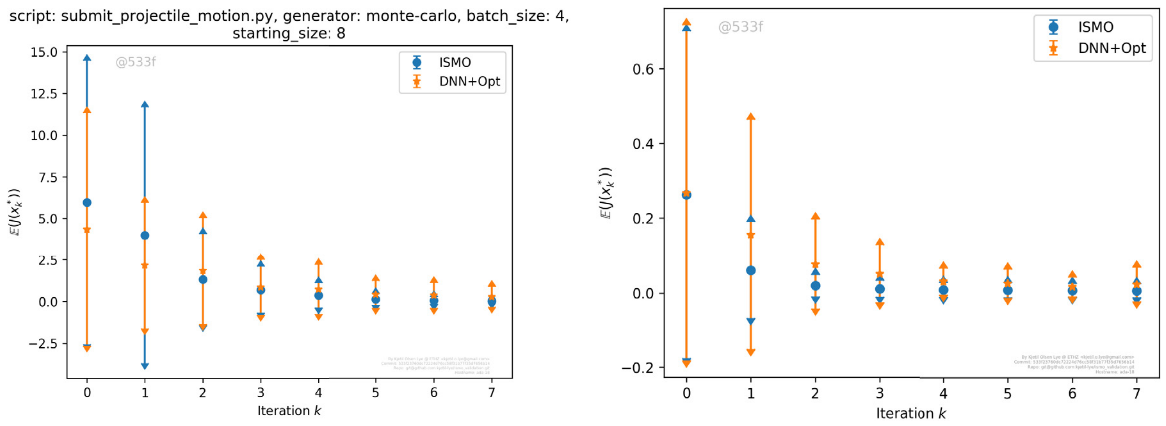


Fig. 4. The objective function vs. number of iterations of the DNNopt and ISMO algorithms for the optimal control of projectile motion. The mean of the objective function and mean ± standard deviation (3.16) are shown. Left: Random initial training points with $N_0 = 8$ and $\Delta N = 4$ for the ISMO algorithm. Right: Sobol initial training points for the ISMO algorithm with $N_0 = 32$ and $\Delta N = 16$.

optimization Algorithm 2.1) for augmenting training points, that are more clustered around the underlying minima, in each successive iteration. This is illustrated in Fig. 2, where we plot the training set $\mathbb{S}_k$ and the additional training points $\bar{\mathbb{S}}_k$ at different iterations k for the ISMO algorithm. As observed in this figure, we start with a randomly chosen set of points, which have no information on the minimizers of the underlying function (the parabola in Fig. 2). At the very first iteration, the standard optimization algorithm identifies optimizers that are clustered near the parabola and at each successive step, more such training points are identified, which increasingly improve the accuracy of the deep neural network around minima and hence, improve the accuracy and robustness of the ISMO algorithm.

We recall from the theory presented in Lemma 4.3 that the initial number of training points N_0 in the ISMO algorithm should be sufficiently high in order to enforce (4.7) and obtain the desired rate of convergence. To check whether a minimum number of initial training points is necessary for the ISMO algorithm to be robust and accurate, we set $N_0 = 8$ and $\Delta N = 4$ in Algorithm 4.1 and present the results in Fig. 4 (left). As seen from this figure and contrasting the results with those presented in Fig. 3, we observe that the ISMO algorithm is actually *less robust* than the DNNopt algorithm, at least for the first few iterations. Hence, we clearly need enough initial training samples for the ISMO algorithm to be robust and efficient.

Finally, the results presented so far used randomly chosen initial training points for the ISMO algorithm. The results with low-discrepancy Sobol points are presented in Fig. 4 (right) and show the same qualitative behavior as those with randomly chosen points. Although the mean and standard deviation of the objective function are slightly smaller with the Sobol points, the differences are minor and we will only use randomly chosen initial training points for the rest of the paper.

This numerical experiment is openly available at https://github.com/kjetil-lye/ismo_validation.

5.3. Inverse problem for the heat equation

In this section, we consider our first PDE example, the well-known heat equation in one space dimension, given by,

$$\begin{aligned} u_t &= q u_{xx}, \quad (x,t) \in (0,1) \times (0,T), \\ u(x,0) &= \bar{u}(x), \quad x \in (0,1), \\ u(0,t) &= u(1,t) \equiv 0, \quad t \in (0,T). \end{aligned} \tag{5.4}$$

Here, u is the temperature and $q \geq 0$ is the diffusion coefficient. We will follow [40] and identify parameters for the following *data assimilation problem*, i.e., for an initial data of the form,

$$\bar{u}(x) = \sum_{\ell=1}^{L} a_\ell \sin(\ell \pi x), \tag{5.5}$$

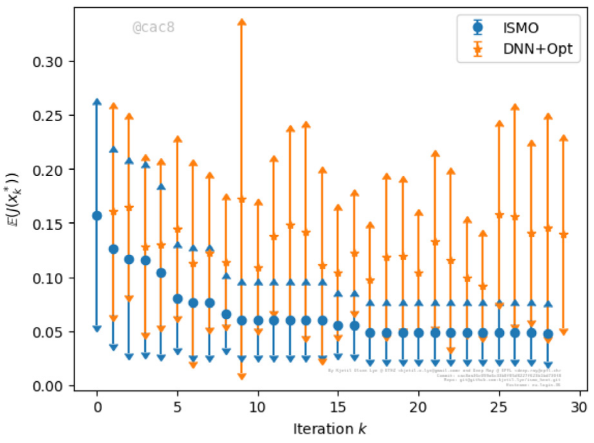


Fig. 5. The objective function (5.7) vs. number of iterations of the DNNopt and ISMO algorithms for the parameter identification (data assimilation) problem for the heat equation. The mean of the objective function and mean ± standard deviation (3.16) are shown.

with unknown coefficients a_ℓ and for an unknown diffusion coefficient q, the aim of this data assimilation or parameter identification problem is to compute approximations to a_ℓ, q from measurements of the form $u_i = u(x_i, T)$, i.e., pointwise measurements of the solution field at the final time.

This problem can readily be cast in the framework of PDE constrained optimization, Section 2, in the following manner. The control parameters are $y = [q, a_1, a_2, \dots, a_L] \in Y = [0, 1]^{L+1}$ and the relevant observables are,

$$\mathcal{L}_i(y) = u(x_i, T), \quad 1 \leqslant i \leqslant I. \tag{5.6}$$

Following [40] and references therein, we can set up the following cost function,

$$\mathcal{G}(y) := \frac{1}{2I} \sum_{i=1}^{I} |\mathcal{L}_i(y) - \bar{\mathcal{L}}_i|^2, \tag{5.7}$$

with measurements $\bar{\mathcal{L}}_i$. In this article, we assume that the measurements are noise free.

In order to perform this experiment, we set $L = 9$, $I = 5$ and $T = 0.001$ and generate the measurement data from the following *ground truth* parameters,

$$[q, a_1, \dots, a_9] = [0.75, 0.1, 0.4, 0.8, 0.25, 0.75, 0.45, 0.9, 0.25, 0.85],$$

and measurement points,

$$[x_1, \dots, x_5] = [0.125, 0.25, 0.5, 0.75, 0.825]$$

We numerically approximate the heat equation (5.4) with a second-order Crank–Nicolson finite difference scheme with 2048 points in both space and time and generate the measurement data $\bar{\mathcal{L}}_i$, for $1 \leqslant i \leqslant 5$ from this simulation.

DNNopt Algorithm 3.1 is run for this particular configuration with the hyperparameters in Table 1. We would like to point out that I deep neural networks, $\mathcal{L}_i^*$, for $1 \leqslant i \leqslant I$, are independently trained, corresponding to each of the observables $\mathcal{L}_i$. The Algorithm 3.1 and its analysis in Section 3.1, can be readily extended to this case.

Results of the DNNopt algorithm with randomly chosen N_k training points, with $N_k = N_0 + k\Delta N$, with $N_0 = 64$ and $\Delta N = 16$, and with $M_k = N_k$ in Algorithm 3.1 in order to compute sensitivity statistics, are presented in Fig. 5. We observe from this figure that the mean $\bar{\mathcal{G}}$ of the cost function (3.16) barely decays with increasing number of training samples. Moreover, the standard deviation (3.16) is also unacceptably high, even for a very high number of training samples. Comparing with the reasonably good performance of the DNNopt algorithm in the last numerical experiment (which had two dimensions), the inadequate performance on this 10-dimensional experiment suggests that DNNopt suffers from a possible curse of dimensionality.

Next, ISMO Algorithm 4.1 is run on this problem with the same setup as for the DNNopt algorithm, i.e., $I = 5$ independent neural networks $\mathcal{L}_{k,i}^*$ are trained, each corresponding to the observable $\mathcal{L}_i$ (5.6), at every iteration k of the ISMO algorithm. The results, with $N_0 = 64$ and $\Delta N = 16$ as hyperparameters of the ISMO algorithm are

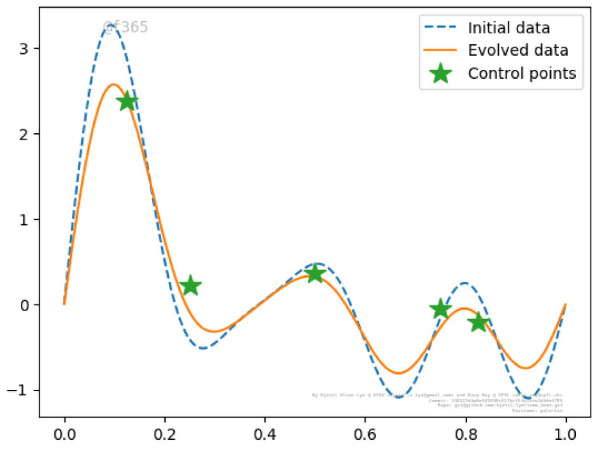


Fig. 6. Solution of the data assimilation problem for the heat equation with the ISMO algorithm. The identified initial data (5.5), resulting (exact) solution at final time and measurements (5.6) are shown.

shown in Fig. 5. We observe from this figure that the mean $\bar{\mathcal{G}}$ of the cost function (3.16) decays with increasing number of iterations, till a local minimum corresponding to a mean objective function value of approximately 0.05 is reached. The standard deviation also decays as the number of iterations increases. In particular, the ISMO algorithm provides a significantly lower mean value of the cost function compared to the DNNopt algorithm. Moreover, the standard deviation is several times lower for the ISMO algorithm than for the DNNopt algorithm. Thus, we expect a robust parameter identification with the ISMO algorithm. This is indeed verified from Fig. 6, where we plot the identified initial condition (5.5) with the ISMO algorithm with a starting size of 64 and a batch size of 16, and the corresponding solution field at time T (computed with the Crank–Nicolson scheme), superposed with the measurements. We see from this figure that the identified initial data leads to one of the multiple solutions of this ill-posed inverse problem that approximates the measured values quite accurately.

This numerical experiment is openly available at https://github.com/kjetil-lye/ismo_heat.

5.4. Shape optimization of airfoils

For the final numerical example in this paper, we choose optimization of the shapes of airfoils (wing cross-sections) in order to improve certain aerodynamic properties. The flow past the airfoil is modeled by the two-dimensional Euler equations, given by,

$$\mathbf{U}_t + \mathbf{F}^x(\mathbf{U})_x + \mathbf{F}^y(\mathbf{U})_y = 0, \quad \mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix}, \quad \mathbf{F}^x(\mathbf{U}) = \begin{pmatrix} \rho u \\ \rho v^2 + p \\ \rho u v \\ (E+p)u \end{pmatrix}, \quad \mathbf{F}^y(\mathbf{U}) = \begin{pmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ (E+p)v \end{pmatrix} \tag{5.8}$$

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 $a = \sqrt{\gamma p/\rho}$ and the Mach number $M = |\mathbf{u}|/a$.

We will follow standard practice in aerodynamic shape optimization [4] and consider a reference airfoil with upper and lower surfaces of the airfoil are located at $(x, y^{U,ref}(x/c))$ and $(x, y^{L,ref}(x/c))$, with $x \in [0, c]$, c being the airfoil chord, and $y^{U,ref}$, $y^{L,ref}$ corresponding to the well-known RAE2822 airfoil [41], see Fig. 7 for a graphical representation of the reference airfoil.

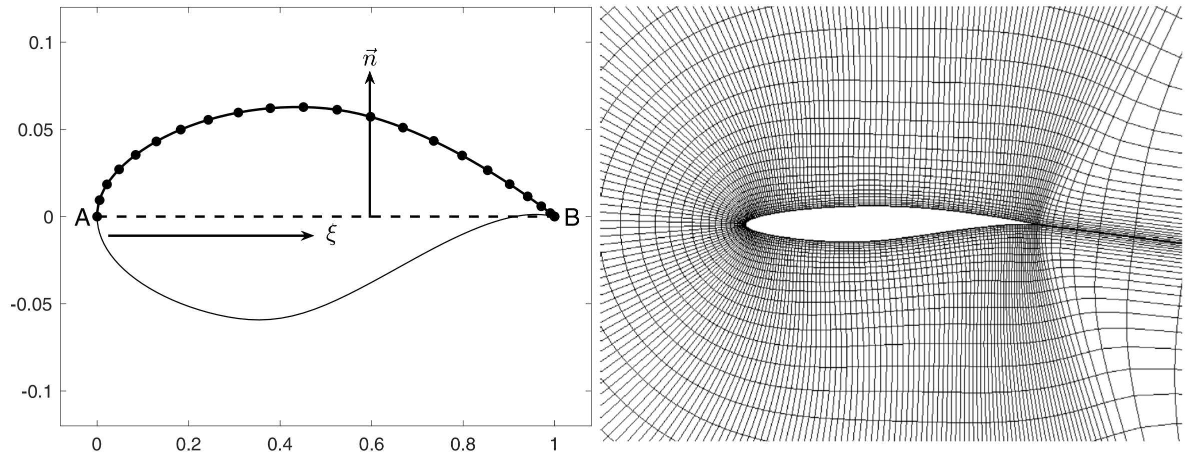


Fig. 7. The reference shape of the RAE2822 airfoil. Left: The parametrization used to deform the shape of the airfoil according to (5.9). Right: The initial C-grid mesh used around the airfoil.

The shape of this reference airfoil is perturbed in order to optimize aerodynamic properties. The upper and lower surface of the airfoil are perturbed as

$$y = y^{ref}(\xi) + h(\xi), \quad h(\xi) = \sum_{k=1}^{d/2} a_k B_k(\xi), \qquad \xi = x/c, \tag{5.9}$$

with c being the airfoil chord length i.e., $c = |AB|$ for the airfoil depicted in Fig. 7.

Although several different parametrizations of shape or its perturbations [42,43] are proposed in the large body of aerodynamic shape optimization literature, we will focus on the simple and commonly used *Hicks–Henne* bump functions [43] to parametrize the perturbed airfoil shape

$$B_k(\xi) = \sin^3(\pi \xi^{q_k}), \quad q_k = \frac{\ln 2}{\ln(d+4) - \ln k}. \tag{5.10}$$

Furthermore, the design parameters for each surface is chosen as

$$\begin{aligned} a_k^U &= 2(y_k^U - 0.5)\left(\frac{d}{2} - k + 1\right) \times 10^{-3}, \quad y_k^U \in [0,1], \\ a_k^L &= 2(y_k^L - 0.5)\,(k+1) \times 10^{-3}, \quad y_k^L \in [0,1], \end{aligned} \tag{5.11}$$

for $1 \leqslant k \leqslant d/2$. The transform functions in (5.11) to obtain the design parameters have been chosen to ensure (i) the perturbation is smaller near narrower tip of the airfoil (point B), and (ii) the design envelope is large enough (see Fig. 8) while avoiding non-physical intersections of the top and bottom surfaces after deformation. Note that the airfoil shape is completely parametrized by the vector $y \in [0,1]^d$. For our experiments, we choose 10 parameters for each surface, i.e., we set $d = 20$.

The flow around the airfoil is characterized by the free-stream boundary conditions corresponding to a Mach number of $M^\infty = 0.729$ and angle of attack of $\alpha = 2.31°$. The observables of interest are the lift and drag coefficients given by,

$$C_L(y) = \frac{1}{K^\infty(y)} \int_S p(y) n(y) \cdot \hat{y}\, ds, \tag{5.12}$$

$$C_D(y) = \frac{1}{K^\infty(y)} \int_S p(y) n(y) \cdot \hat{x}\, ds, \tag{5.13}$$

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 $\hat{x} = [\cos(\alpha), \sin(\alpha)]$.

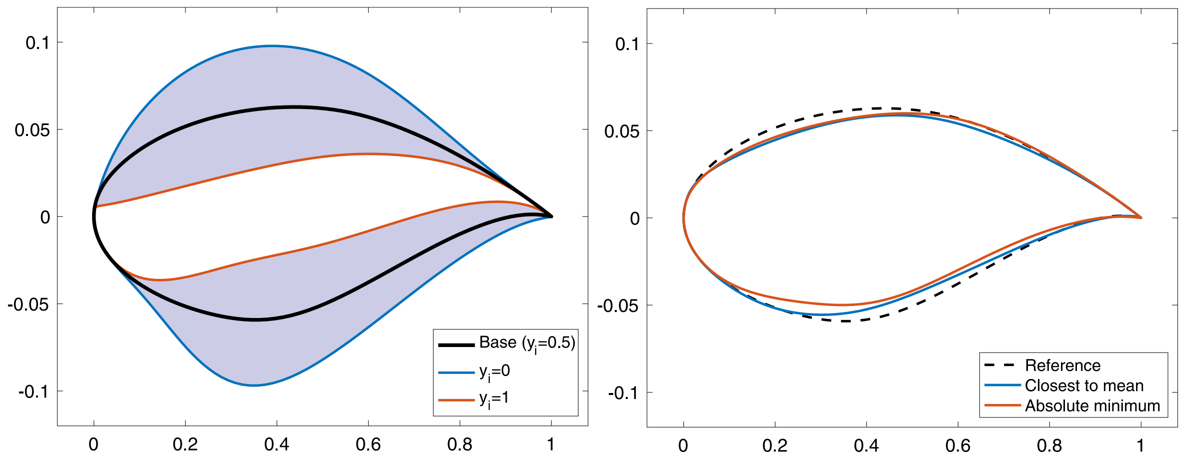


Fig. 8. Shapes for the airfoil shape optimization problem. Left: The design envelope, where the reference, widest and narrowest airfoils are obtained by setting all design parameters y_i as 0.5, 0 and 1, respectively. Right: The optimized shapes obtained using the ISMO algorithm.

As is standard in aerodynamic shape optimization [3,4,44], we will modulate airfoil shape in order to *minimize drag while keeping lift constant*. This leads to the following optimization problem,

$$\mathcal{G}(y) = \frac{C_D(y)}{C_D^{ref}} + P \max\left(0, 0.999 - \frac{C_L}{C_L^{ref}}\right) \tag{5.14}$$

with $C_{L,D}$ being the lift (5.12) and drag (5.13) coefficients, respectively, and C_L^{ref} is the lift corresponding to the reference RAE2822 airfoil and $P = 10\,000$ is a parameter to penalize any deviations from the lift of the reference airfoil. The reference lift is calculated to be $C_L^{ref} = 0.8763$, and the reference drag is $C_D^{ref} = 0.011562$.

We will compute approximate minimizers of the cost function (5.14) with the DNNopt Algorithm 3.1 and the ISMO Algorithm 4.1. The training data is generated using the NUWTUN solver[1] to approximate solutions of the Euler equations (5.8) around the deformed airfoil geometry (5.9), with afore-mentioned boundary conditions. The NUWTUN code is based on the ISAAC code[2] [45] and employs a finite volume solver using the Roe numerical flux and second-order MUSCL reconstruction with the Hemker–Koren limiter. Once the airfoil shape is perturbed, the base mesh (see Fig. 7) is deformed to fit the new geometry using a thin plate splines based radial basis function interpolation technique [46,47], which is known to generate high quality grids. The problem is solved to steady state using an implicit Euler time integration.

First, we run the DNNopt Algorithm 3.1, with two independent neural networks C_L^* and C_D^* for the lift coefficient C_L and drag coefficient C_D, respectively. The results of the DNNopt algorithm, with randomly chosen training points in $Y = [0, 1]^{20}$, and with $N_0 + k\Delta N$ training samples, for $N_0 = 64$ and $\Delta N = 16$ and with $M_k = N_k$ in Algorithm 3.1 in order to compute sensitivity statistics, are presented in Fig. 9. From this figure, we observe that the mean $\bar{\mathcal{G}}$ of the cost function (5.14) decays, but in an oscillatory manner, with increasing number of training samples. Moreover, the standard deviation (3.16) also decays but is still rather high, even for a large number of iterations. This is consistent with the findings in the other two numerical experiments reported here. These results suggest that the DNNopt algorithm may not yield robust or accurate optimal shapes.

Next, we run ISMO Algorithm 4.1 with the same setup as DNNopt, i.e., with two independent neural networks $C_{L,k}^*, C_{D,k}^*$, at every iteration k, for the lift and the drag, respectively. We set hyperparameters $N_0 = 64$ and batch size $\Delta N = 16$ and present results with the ISMO algorithm in Fig. 9. We see from this figure that the mean and the standard deviation of the cost function steadily decay with increasing number of iterations for the ISMO algorithm. Moreover, the mean as well as the standard deviation of cost function, approximated by the ISMO algorithm, is significantly less than those by the DNNopt algorithm. Thus, one can expect that the ISMO algorithm provides

[1] http://bitbucket.org/cpraveen/nuwtun.
[2] http://isaac-cfd.sourceforge.net.

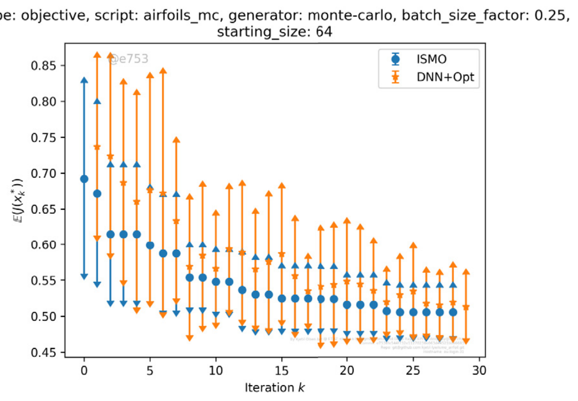
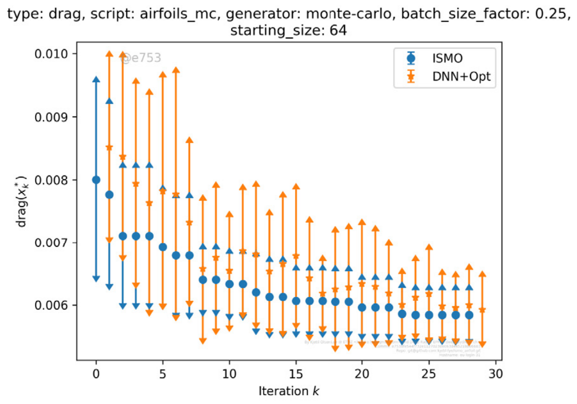


Fig. 9. The objective function (5.14) vs. number of iterations of the DNNopt and ISMO algorithms for the airfoil shape optimization problem. The mean of the objective function and mean ± standard deviation (3.16) are shown.



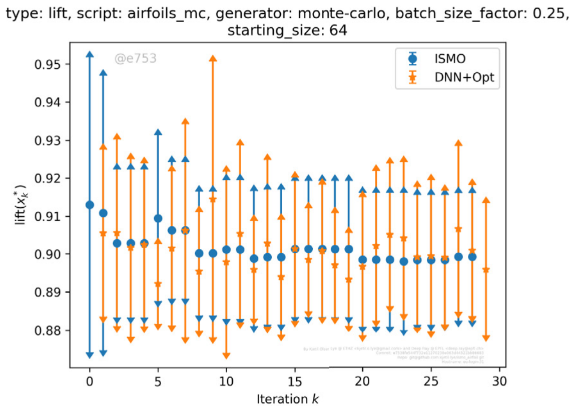


Fig. 10. The Drag (5.13) (Left) and Lift (5.12) (Right) vs. number of iterations of the DNNopt and ISMO algorithms for the airfoil shape optimization problem. The mean of the objective function and mean ± standard deviation (3.16) are shown.

parameters for robust shape optimization of the airfoil. This is further verified in Fig. 10, where we separately plot the mean and mean ± standard deviation for the drag and lift coefficients, corresponding to the optimizers at the each iteration, for the DNNopt and ISMO algorithms. From this figure, we observe that the mean and standard deviation for the computed drag, with the ISMO algorithm, decay with increasing number of iterations. Moreover, both the mean and standard deviation of the drag are significantly lower than those computed with the DNNopt algorithm. On the other hand, both algorithms result in Hicks–Henne parameters that keep lift nearly constant (both in mean as well as in mean ± standard deviation), around a value of $C_L = 0.90$, which is slightly higher than the reference lift. This behavior is completely consistent with the structure of the cost function (5.14), which places the onus on minimizing the drag, while keeping lift nearly constant.

Quantitatively, the ISMO algorithm converges to a set of parameters (see the resulting shape in Fig. 8) that correspond to the computed minimum drag coefficient of $C_D = 0.005202$, when compared to the drag $C_D^{ref} = 0.01156$, of the reference RAE2822 airfoil. The corresponding lift coefficient with this optimizer is $C_L = 0.8960$, when compared to the reference lift $C_L^{ref} = 0.8763$. Similarly, the computed Hicks–Henne parameters, with the ISMO algorithm, that lead to a cost function (5.14) that is closest to the mean cost function $\bar{\mathcal{G}}$ (see shape of the resulting airfoil in Fig. 8), resulted in a drag coefficient of $C_D = 0.005828$ and lift coefficient of $C_L = 0.8879$. Thus, while the optimized lift was kept very close to the reference lift, the drag was reduced by approximately 55% (for the absolute optimizer) and 50% (on an average) by the shape parameters, computed with the ISMO algorithm.

In order to investigate the reasons behind this significant drag reduction, we plot the Mach number around the airfoil to visualize the flow, for the reference airfoil and two of the optimized airfoils (one corresponding to the

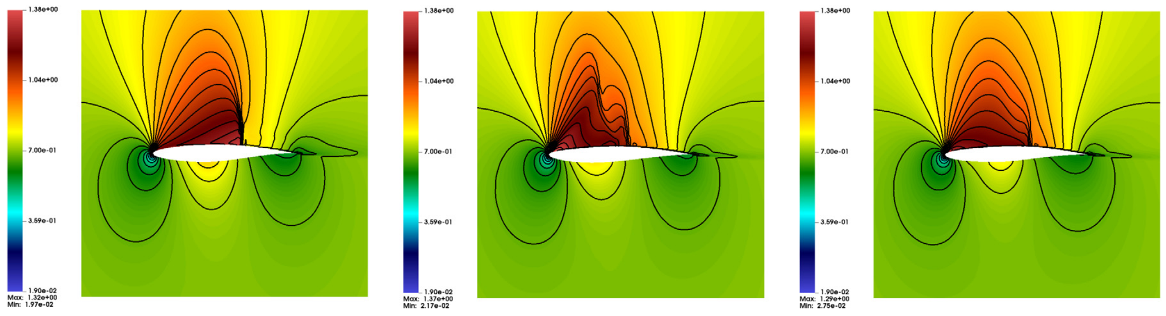


Fig. 11. Mach number contours for the flow around airfoils. Left: Flow around reference RAE2822 airfoil. Center: Flow around optimized airfoil, corresponding to Hicks–Henne parameters with a cost function (5.14) that is closest of the (converged) mean cost function for the ISMO algorithm. Right: Flow around optimized airfoil, corresponding to Hicks–Henne parameters with the smallest (converged) cost function for the ISMO algorithm.

absolute smallest drag and the other to the mean optimized drag) in Fig. 11. From this figure we see that the key difference between the reference and optimized airfoil is significant reduction (even elimination) in the strength of the shock over the upper surface of the airfoil. This diminishing of shock strength reduces the shock drag and the overall cost function (5.14).

Finally, we compare the performance of the ISMO algorithm for this example with a standard optimization algorithm. To this end, we choose a black box truncated Newton (TNC) algorithm, with its default implementation in *SciPy* [38]. In Fig. 12, we plot the mean and the mean $\pm$ standard deviation of the cost function with the TNC algorithm, with 1024 starting values chosen randomly and compare it with ISMO Algorithm 4.1. From Fig. 12 (left), we observe that the ISMO algorithm has a significantly lower mean of the objective function, than the black box TNC algorithm, even for a very large number (2^{11}–2^{12}) of evaluations (calls) to the underlying PDE solver. Clearly, the ISMO algorithm readily outperforms the TNC algorithm by more than an order of magnitude, with respect to the mean of the objective function (5.14). Moreover, we also observe from Fig. 12 (right) that the ISMO algorithm has significantly (several orders of magnitude) lower standard deviation than the TNC algorithm for the same number of calls to the PDE solver. Thus, this example clearly illustrates that ISMO algorithm is both more efficient as well as more robust than standard optimization algorithms such as the truncated Newton methods.

This numerical experiment is openly avaiable at https://github.com/kjetil-lye/ismo_airfoil.

6. Discussion

Robust and accurate numerical approximation of the solutions of PDE constrained optimization problems represents a formidable computational challenge as existing methods might require a large number of evaluations of the solution (and its gradients) for the underlying PDE. Since each call to the PDE solver is computationally costly, multiple calls can be prohibitively expensive.

Using surrogate models within the context of PDE constrained optimization is an attractive proposition. As long as the surrogate provides a robust and accurate approximation to the underlying PDE solution, while being much cheaper to evaluate computationally, surrogate models can be used inside standard optimization based algorithms, to significantly reduce the computational cost. However, finding surrogates with desirable properties can be challenging for high dimensional problems.

Deep neural networks (DNNs) have emerged as efficient surrogates for PDEs, particularly for approximating *observables* of PDEs, see [27–29] and references therein. Thus, it is natural to investigate whether DNNs can serve as efficient surrogates in the context of PDE constrained optimization. To this end, we propose DNNopt Algorithm 3.1 that combines standard optimization algorithms, such as quasi-Newton Algorithm 2.1, with deep neural network surrogates.

However, numerical examples presented in this paper show that the DNNopt algorithm converges slowly, particularly for high-dimensional problems. Moreover, it is found to be sensitive to the choice of initial starting values for the optimization algorithm. These deficiencies of the DNNopt algorithm were also suggested by theoretical analysis, albeit in a restrictive setting with stringent hypothesis on the underlying problem and approximating neural networks.

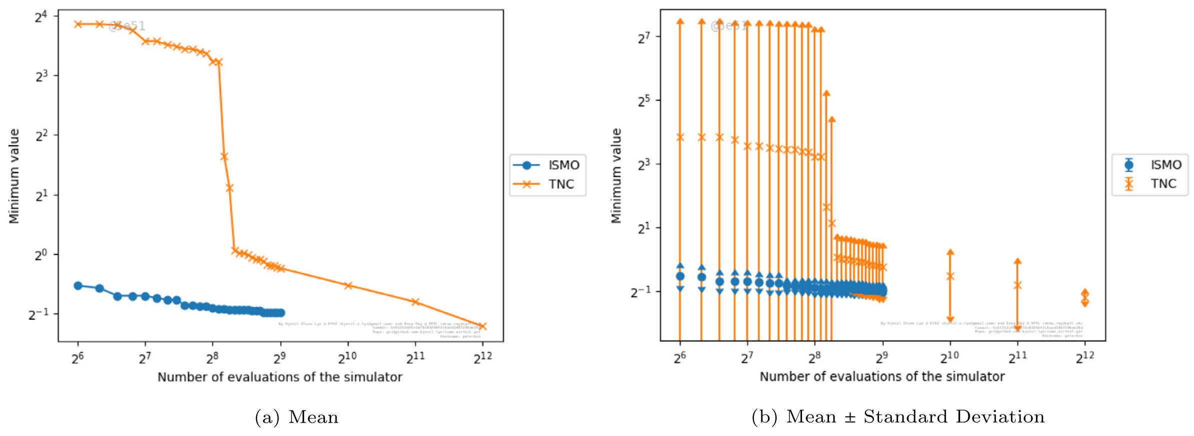


Fig. 12. Comparison of ISMO with the black box TNC algorithm of [38] for the airfoil shape optimization example. Left: Mean of the cost function (5.14) vs. number of calls to the PDE solver. Right: Mean ± Standard deviation (3.16) for the cost function (5.14).

We find that fixing training sets *a priori* for the deep neural networks might lead to poor approximation of the underlying minima for the optimization problem, which correspond to a subset (manifold) of the parameter space. To alleviate this shortcoming of the DNNopt algorithm, we propose a novel algorithm termed as iterative surrogate model optimization (ISMO) Algorithm 4.1. The key idea behind ISMO is to iteratively augment the training set for a sequence of deep neural networks such that the underlying minima of the optimization problem are better approximated. At any stage of the ISMO iteration, the current state of the DNN is used as an input to the standard optimization algorithm to find (approximate) local minima for the underlying cost function. These local minima are added to the training set and independent DNNs for the next step are trained on the augmented set. This feedback algorithm is iterated till convergence. Thus, one can view ISMO as an *active learning algorithm* [30], where the learner (the deep neural network) queries the teacher or oracle (standard optimization Algorithm 2.1) to provide a better training set at each iteration.

We tested the ISMO and DNNopt algorithms for three representative numerical examples, namely for an optimal control problem for a nonlinear ODE, a data assimilation (parameter identification) inverse problem for the heat equation and shape optimization of airfoils, subject to the Euler equations of compressible fluid dynamics. For all these examples, the ISMO algorithm was shown to significantly outperform the DNNopt algorithm, both in terms of the decay of the (mean) objective function and its greatly reduced sensitivity to starting values. Moreover, the ISMO algorithm outperformed (by more than an order of magnitude) a standard black-box optimization algorithm for aerodynamic shape optimization.

Thus, we can assert that the active learning ISMO algorithm provides a very promising framework for significantly reducing the computational cost of PDE constrained optimization problems, while being robust and accurate. Moreover, it is very flexible, with respect to the choice of the underlying optimization algorithm as well as architecture of neural networks, and is straightforward to implement in different settings.

This article is the first to present the ISMO algorithm and readily lends itself to the following extensions.

- We have presented the ISMO algorithm in a very general setting of an abstract PDE constrained optimization problem. Although we have selected three representative examples in this article, the algorithm itself can be applied in a straightforward manner to a variety of PDEs, both linear and non-linear.
- We have focused on a particular type of underlying optimization algorithm, i.e., of the quasi-Newton type. However, ISMO Algorithm 4.1 does not rely on any specific details of the optimization algorithm. Hence, any optimization algorithm can be used as the teacher or oracle within ISMO. In particular, one can also use gradient-free optimization algorithms such as particle swarm optimization and genetic algorithms within ISMO. Thus, even black-box optimization algorithms can employed within ISMO to significantly accelerate finding solutions of PDE constrained optimization problems.
- Similarly, the ISMO algorithm does not rely on a specific structure of the surrogate model. Although we concentrated on neural networks in this article, one can readily use other surrogates such as Gaussian process regression, within the ISMO algorithm.

- In this article, we have considered the PDE constrained optimization problem, where the objective function in (2.3), is defined in terms of an *observable* (2.2) of the PDE solution. In some problems, particularly for control problems, it might be necessary to consider the whole solution field and approximate it with a neural network. Such DNNs are also available but might be more expensive to train and evaluate. Thus, one needs to carefully investigate the resulting speedup with the ISMO algorithm over standard optimization algorithms in this context.
- A crucial issue in PDE constrained optimization is that of optimization under uncertainty, see [8] and references therein. In this context, the ISMO algorithm demonstrates considerable potential to significantly outperform state of the art algorithms and we consider this extension in a forthcoming paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research of SM is partially supported by ERC CoG, Switzerland 770880 SPARCCLE. The research of PC is supported by Department of Atomic Energy, Government of India, under Project No. 12-R&D-TFR-5.01-0520.

References

[1] A. Borzi, V. Schultz, Computational Optimization of Systems Governed by Partial Differential Equations, SIAM, 2012.

[2] F. Troltzsch, Optimal Control of Partial Differential Equations, AMS, 2010.

[3] J. Reuther, A. Jameson, Aerodynamic shape optimization of wing and wing-body configurations using control theory, in: 33rd Aerospace Sciences Meeting and Exhibit, American Institute of Aeronautics and Astronautics, 1995.

[4] B. Mohammadi, O. Pironneau, Applied Shape Optimization for Fluids, Oxford University Press, 2009.

[5] R. Fletcher, Practical Methods of Optimization, John Wiley and sons, 1987.

[6] M. Clerc, Particle Swarm Optimization, Wiley, 2005.

[7] S.N. Sivanandam, S.N. Deepa, Introduction to Genetic Algorithms, Springer, 2008.

[8] C. Schillings, S. Schmidt, V. Schulz, Efficient shape optimization for certain and uncertain aerodynamic design, Comput. & Fluids 46 (2011) 78–87.

[9] A.I.J. Forrester, A. Sóbester, A.J. Keane, Engineering Design via Surrogate Modelling: A Practical Guide, Wiley, 2008.

[10] A. Quarteroni, A. Manzoni, F. Negri, Reduced Basis Methods for Partial Differential Equations: An Introduction, Vol. 92, Springer, 2015.

[11] C.E. Rasmussen, Gaussian processes in machine learning, in: Summer School on Machine Learning, Springer, 2003, pp. 63–71.

[12] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT press, 2016.

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

[14] 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, Annu. Rev. Biochem. 77 (6) (2018) 363–382.

[15] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Netw. 9 (5) (1998) 987–1000.

[16] M. Raissi, G.E. Karniadakis, Hidden physics models: Machine learning of nonlinear partial differential equations, J. Comput. Phys. 357 (2018) 125–141.

[17] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, J. Comput. Phys. 378 (2019) 686–707.

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

[19] S. Mishra, R. Molinaro, Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs, 2020, Preprint, available from arXiv:2006:16144v1.

[20] S. Mishra, R. Molinaro, Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs II: A class of inverse problems, 2020, Preprint, available from ETH Zürich SAM Reports.

[21] J. Han, A. Jentzen, W. E, Solving high-dimensional partial differential equations using deep learning, Proc. Natl. Acad. Sci. 115 (34) (2018) 8505–8510.

[22] W. E, J. Han, 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.

[23] C. Beck, S. Becker, P. Grohs, N. Jaafari, A. Jentzen, Solving stochastic differential equations and Kolmogorov equations by means of deep learning. Preprint, available as arXiv:1806.00421v1.

[24] J. Tompson, K. Schlachter, P. Sprechmann, K. Perlin, Accelerating eulerian fluid simulation with convolutional networks, in: Proceedings of the 34th International Conference on Machine Learning, ICML, Vol. 70, 2017, pp. 3424–3433.

[25] D. Ray, J.S. Hesthaven, An artificial neural network as a troubled-cell indicator, J. Comput. Phys. 367 (2018) 166–191.
[26] S. Mishra, A machine learning framework for data driven acceleration of computations of differential equations, Math. Eng. 1 (2019) 118.
[27] K.O. Lye, S. Mishra, D. Ray, Deep learning observables in computational fluid dynamics, J. Comput. Phys. (2020) 109339.
[28] K.O. Lye, S. Mishra, R. Molinaro, A multi-level procedure for enhancing accuracy of machine learning algorithms, European J. Appl. Math. (2020).
[29] S. Mishra, T.K. Rusch, Enhancing accuracy of deep learning algorithms by training with low-discrepancy sequences, 2020, Preprint, available as arXiv:2005.12564.
[30] B. Settles, Active Learning: Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool, 2012.
[31] R.E. Caflisch, Monte carlo and quasi-monte carlo methods, Acta Numer. 7 (1998) 1–49.
[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, World Scientific, 2005, pp. 49–74.
[33] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, 2015.
[34] F. Cucker, S. Smale, On the mathematical foundations of learning, Bull. Amer. Math. Soc. 39 (1) (2002) 1–49.
[35] J. Calder., Consistency of lipschitz learning with infinite unlabeled data and finite labeled data, SIAM J. Math. Data Sci. 1 (2019) 780–812.
[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org.
[37] F. Chollet, et al., Keras, 2015, https://keras.io.
[38] P. Virtanen, R. Gommers, T.E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S.J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A.R.J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E.W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E.A. Quintero, C.R. Harris, A.M. Archibald, A.H. Ribeiro, F. Pedregosa, P. van Mulbregt, S... Contributors, SciPy 1.0: Fundamental algorithms for scientific computing in python, Nature Methods 17 (2020) 261–272.
[39] IBM, IBM spectrum LDF, 2020, (accessed July 27, 2020).
[40] R. Becker, D. Meidner, B. Vexler, Efficient numerical solution of parabolic optimization problems by finite element methods, Optim. Methods Sofw. (2007).
[41] C. Hirsch, D. Wunsch, J. Szumbarski, J. Pons-Prats, et al., Uncertainty Management for Robust Industrial Design in Aeronautics, in: Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Springer, 2019.
[42] J.A. Samareh, Survey of shape parameterization techniques for high-fidelity multidisciplinary shape optimization, AIAA J. 39 (5) (2001) 877–884.
[43] D.A. Masters, N.J. Taylor, T. Rendall, C.B. Allen, D.J. Poole, Geometric comparison of aerofoil shape parameterization methods, AIAA J. (2017) 1575–1589.
[44] Z. Lyu, G.K.W. Kenway, J.R.R.A. Martins, Aerodynamic shape optimization investigations of the common research model wing benchmark, AIAA J. 53 (4) (2015) 968–985.
[45] J.H. Morrison, A Compressible Navier-Stokes Solver with Two-Equation and Reynolds Stress Turbulence Closure Models, NASA Contractor Report 4440, NASA, 1992.
[46] K. Kumar, M.T. Nair, A Mesh Deformation Strategy for Multiblock Structured Grids, Project Document CF 0708, National Aerospace Laboratories, 2007.
[47] R. Duvigneau, P. Chandrashekar, Kriging-based optimization applied to flow control, Internat. J. Numer. Methods Fluids 69 (11) (2011) 1701–1714.