Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 398 (2022) 115284
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

Numerical approximation of partial differential equations by a variable projection method with artificial neural networks

Suchuan Dong*, Jielin Yang

Center for Computational and Applied Mathematics, Department of Mathematics, Purdue University, USA

Received 8 February 2022; received in revised form 25 May 2022; accepted 23 June 2022
Available online xxxx

Abstract

We present a method for solving linear and nonlinear partial differential equations (PDE) based on the variable projection framework and artificial neural networks. For linear PDEs, enforcing the boundary/initial value problem on the collocation points gives rise to a separable nonlinear least squares problem about the network coefficients. We reformulate this problem by the variable projection approach to eliminate the linear output-layer coefficients, leading to a reduced problem about the hidden-layer coefficients only. The reduced problem is solved first by the nonlinear least squares method to determine the hidden-layer coefficients, and then the output-layer coefficients are computed by the linear least squares method. For nonlinear PDEs, enforcing the boundary/initial value problem on the collocation points gives rise to a nonlinear least squares problem that is not separable, which precludes the variable projection strategy for such problems. To enable the variable projection approach for nonlinear PDEs, we first linearize the problem with a Newton iteration, using a particular linearization formulated in terms of the updated approximation field. The linearized system is solved by the variable projection framework together with artificial neural networks. Upon convergence of the Newton iteration, the neural-network coefficients provide the representation of the solution field to the original nonlinear problem. We present ample numerical examples to demonstrate the performance of the method developed herein. For smooth field solutions, the errors of the current method decrease exponentially as the number of collocation points or the number of output-layer coefficients increases. We compare extensively the current method with the extreme learning machine (ELM) method from a previous work. Under identical conditions and configurations, the current method exhibits an accuracy significantly superior to the ELM method.

Keywords: Artificial neural networks; Variable projection; Linear least squares; Nonlinear least squares; Scientific machine learning; Deep learning

1. Introduction

This work concerns the numerical approximation of partial differential equations (PDE) with artificial neural networks (ANN), and we exploit the variable projection (VarPro) approach [1] together with ANNs for solving linear and nonlinear PDEs. Neural network-based PDE methods, especially those based on deep neural networks (DNN) and deep learning [2], have flourished in the past few years; see e.g. [3–17], and the recent review [18] and the references therein. The DNN-based PDE solvers are fairly straightforward to implement, by encoding the PDEs, the boundary and initial conditions into a cost function and then using some flavor of gradient descent (or back

* Corresponding author.
E-mail address: sdong@purdue.edu (S. Dong).

https://doi.org/10.1016/j.cma.2022.115284
0045-7825/

propagation) type optimization algorithms to minimize this cost. Their weakness lies in the limited accuracy and the high computational cost (long network-training time). Improvements to the neural network training have been considered in a number of recent studies (see e.g. [11,14,19]). Another promising type of neural network-based methods for computational PDEs has recently appeared [20–25], which are based on a type of randomized neural networks called extreme learning machines (ELM) [26,27]. With these methods the weight/bias coefficients in the hidden layers are randomly assigned and fixed. Only the coefficients of the linear output layer are trainable, and they are trained by a linear least squares method for linear PDEs and by a nonlinear least squares method for nonlinear PDEs [21]. It has been shown in [21] that the accuracy and the computational cost (network training time) of the ELM-based method are advantageous compared with those of the DNN-based PDE solvers. In addition, the computational performance of the ELM-type method from [21] is observed to be comparable to or exceed that of the classical finite element method (FEM). Further extensions and improvements to the ELM method of [21] have been documented in [25] recently, which compares systematically the improved ELM with the classical and high-order FEM for solving a number of problems. The improved ELM outperforms the classical second-order FEM by a considerable margin, and it outcompetes the high-order FEM when the problem size is not very small [25].

Variable projection (VarPro) is a classical approach for solving separable nonlinear least squares (SNLLS) problems [1,28]. These problems are separable in the sense that the unknown parameters or variables can be separated into two sets: the linear parameters and the nonlinear parameters. Problems of this kind often involve a model function that is a linear combination of parameterized nonlinear basis functions. The basic idea of VarPro is to treat the linear parameters as dependent on the nonlinear parameters, and then eliminate the linear parameters from the problem by using the linear least squares method. This gives rise to a reduced, but generally more complicated, nonlinear least squares problem that involves only the nonlinear parameters [28]. One can then solve the reduced problem for the nonlinear parameters by a nonlinear least squares method, typically involving a Gauss–Newton type algorithm coupled with trust region or backtracking line search strategies [29,30]. Upon attaining the nonlinear parameters, one then computes the linear parameters by the linear least squares method. Although the reduced problem is in general more complicated, the benefits of variable projection are typically very significant. These include the reduced dimension of the parameter space, better conditioning, and faster convergence with the reduced problem [28,31,32]. In some sense the idea of variable projection to least squares problems can be analogized to the Schur complement in linear algebra or the static condensation in computational mechanics (see e.g. [33]).

The VarPro algorithm was originally developed in [1], and has been improved and generalized by a number of researchers and applied to many areas in the past few decades [28,31,34–44]. In [1] the authors have proved the equivalence between the solution of the VarPro reduced formulation and that of the original problem, and developed differentiation formulas for the orthogonal projectors and the Moore–Penrose pseudoinverses, which are critical to the computation of the Jacobian matrix in the nonlinear least squares solution of the reduced problem. An important simplification to the VarPro algorithm is suggested in [34], which involves computing an approximate Jacobian rather than the true Jacobian. This significantly reduces the per-iteration cost of VarPro, with generally insignificant or negligible sacrifice to the accuracy for many problems [45]. The variable projection algorithms for problems with constraints on the linear or nonlinear parameters are investigated in e.g. [38,46–48], among others. The implementations of the VarPro method have been discussed in [38,49]. In [28] the original developers of VarPro have reviewed the developments of this method up to the early 2000s and compiled an extensive list of areas for its ongoing and potential applications. A generalization of the variable projection approach has been considered in [31], which deals with two separate classes of variables without requiring one class to be linear; see also more recent contributions on the generalization of VarPro in e.g. [43,50–52]. We would also like to mention the simplification of the Jacobian matrix in [53], and the algorithm of [54], which resembles the variable projection approach in some sense; see a comparison of these algorithms with the variable projection method in [45]. An approach related to variable projection is the so-called block coordinate descent [55], which alternates between the minimization of two separate sets of variables involved in the problem [31,35,56].

The VarPro algorithm or its variants for training neural networks have been the subject of several studies in the literature [32,44,57–62]. The projection learning algorithm developed in [57–59] is in the same spirit as variable projection, and it computes the linear parameters by the linear least squares method and the nonlinear parameters by a gradient descent scheme. In [32] the authors have proved that the reduced nonlinear functional of the variable projection approach, while seemingly more complicated, leads to a better-conditioned problem and always converges faster than the original problem; see also [31]. The VarPro method together with the Levenberg–Marquardt algorithm

is employed for the training of two-layered neural networks in [60,61] and compared with other related approaches. In the recent works [44,62] the authors extend the variable projection approach to deal with non-quadratic objective functions, such as the cross-entropy function in classification tasks, and also present a stochastic optimization method (termed "slimTrain") based on variable projection for training deep neural networks with attractive properties.

In the current work we focus on the variable projection approach for solving partial differential equations. We numerically approximate the solution fields to linear and nonlinear PDEs by exploiting variable projection together with artificial neural networks. For computational PDEs, the issues one would encounter with VarPro are a little different from those for data fitting problems or function approximations, which account for the majority of VarPro applications in the literature so far. For PDEs, one does not have the data for the field function to be solved for, unlike in data fitting problems. What one does have are the conditions (or constraints) the solution field needs to satisfy, namely, the PDEs, the boundary conditions, and also the initial conditions if the problem is time-dependent. To deal with this type of problems, the variable projection method needs to be adapted accordingly.

The general approach with VarPro and artificial neural networks for solving PDEs is as follows. We employ a feed-forward neural network with one or more hidden layers to represent the field solution to the PDE, requiring that the output layer be linear (i.e. applying no activation function) and with zero bias. We enforce the PDEs on a set of collocation points in the domain, and enforce the boundary/initial conditions on a set of collocation points on the appropriate boundaries of the spatial (or spatial–temporal) domain. This gives rise to a set of discrete equations about the field function to be solved for, which depends on the weight/bias coefficients in the output/hidden layers of the neural network. In turn, this set of equations leads to a nonlinear least squares problem about the neural-network coefficients, providing an opportunity for the variable projection method if this nonlinear least squares problem is separable.

It is necessary to distinguish two types of linearities (or nonlinearities) before the variable projection can be used to solve the above nonlinear least squares problem. The first type concerns whether the network coefficients are linear (or nonlinear) with respect to the output field of the network. Since no activation function is applied to the output layer, the output-layer coefficients are linear and the hidden-layer coefficients are nonlinear with respect to the network output. The second type concerns whether the boundary/initial value problem with the given PDE is linear (or nonlinear) with respect to the field function to be solved for.

If the boundary/initial value problem is linear, i.e. with both linear PDE and linear boundary/initial conditions, then the aforementioned nonlinear least squares problem is separable. The output-layer coefficients of the neural network are the linear parameters and the hidden-layer coefficients are the nonlinear parameters in this separable nonlinear least squares problem. In this case, employing VarPro for training the neural network to solve the given boundary/initial value problem would be conceptually straightforward.

On the other hand, if the boundary/initial value problem is nonlinear, i.e. either the PDE itself or the associated boundary/initial conditions are nonlinear, the aforementioned nonlinear least squares problem is not separable. In this case all the network coefficients become nonlinear parameters in the aforementioned nonlinear least squares problem. Therefore, the variable projection cannot be directly used for solving nonlinear PDEs (or problems with nonlinear boundary/initial conditions). How to enable the variable projection method to solve nonlinear PDEs is the focus of the current work.

In this paper we present a Newton-variable projection method together with artificial neural networks for solving nonlinear boundary/initial value problems (nonlinear PDEs or nonlinear boundary/initial conditions). Given a nonlinear boundary/initial value problem, we first linearize the problem for the Newton iteration, with a particular linearized form. More specifically, the linearization is formulated in terms of the updated approximation field, not the increment field. This linearization form is critical to the accuracy of the current Newton-VarPro method. The linearized system (PDE and boundary/initial conditions) is linear with respect to the updated approximation field, and it is solved by the variable projection approach together with the neural networks. Therefore, to solve nonlinear PDEs, the current method involves an overall Newton iteration. Within each iteration, we use the VarPro method together with ANNs to solve the linearized system to attain the updated field approximation. Upon convergence of the Newton iteration, the neural-network coefficients contain the representation of the solution field to the original nonlinear problem.

The VarPro method together with ANNs for solving linear PDEs has been considered first in this paper. We discuss in some detail how to implement the Jacobian matrix together with neural networks, and how to introduce perturbations in VarPro when solving the reduced problem in order to prevent the solution from being trapped to

the local minima in the nonlinear least squares computation. We have presented several numerical examples, with both linear and nonlinear PDEs, to test the performance of the VarPro method. We observe that, for smooth field solutions, the VarPro errors decrease exponentially as the number of collocation points or the number of output-layer coefficients increases, which is reminiscent of the spectral convergence of traditional high-order methods [33,63–71]. We also compare the performance of the current VarPro method with that of the ELM method from [21,25]. The numerical results show that, under the same conditions and network configurations, VarPro is considerably more accurate than ELM, especially when the size of the neural network is small. On the other hand, the computational cost (i.e. network training time) of VarPro is usually much higher than that of ELM.

In the current work the VarPro method and the neural networks are implemented based on the Tensorflow (www.tensorflow.org) and Keras (keras.io) libraries in Python. The scipy and numpy libraries in Python are used for the linear and nonlinear least squares computations. All the numerical tests are conducted on a MAC computer (Intel Core i5 CPU 3.2 GHz, 24 GB memory) in the authors' institution.

The main contribution of this paper lies in the Newton-VarPro method together with artificial neural networks for solving nonlinear partial differential equations. To the best of the authors' knowledge, this work seems also to be the first time when the variable projection approach (with ANNs) is extended and adapted to solving linear partial differential equations.

The rest of this paper is structured as follows. In Section 2 we first outline how to solve linear PDEs with the variable projection approach together with ANNs. The computations for the reduced residual function and the Jacobian matrix of the reduced problem, and the VarPro algorithm with perturbations are discussed in detail. Then we introduce the Newton-VarPro method together with ANNs for solving nonlinear PDEs. In Section 3 we present several numerical examples with linear and nonlinear PDEs to demonstrate the accuracy of the VarPro method. The performance of the current VarPro method is compared extensively with that of the ELM method from [21,25]. Section 4 then concludes the presentation with some closing remarks and comments on the presented method. In Appendix A we study the effect of random seeds in the random number generator on the VarPro accuracy. Appendix B compares the VarPro with the physics-informed neural network (PINN) method [4]. Appendix C compares the VarPro with the classical finite element method (FEM).

2. Variable projection with artificial neural networks for computational PDEs

We develop an algorithm combining the variable projection (VarPro) framework with artificial neural networks (ANN) for numerically approximating PDEs. For linear PDEs, the ANN representation of the solution field leads to a separable nonlinear least squares problem, which can be solved by the variable projection approach. For nonlinear PDEs, on the other hand, the ANN representation of the solution field leads to a nonlinear least squares (NLLSQ) problem that is not separable, preventing the use of the variable projection strategy. We overcome this issue by a combined Newton-variable projection method, which enables the variable projection approach in solving nonlinear PDEs. In the following subsections we first illustrate the VarPro/ANN algorithm for solving linear PDEs, and then introduce the Newton-VarPro/ANN algorithm for solving nonlinear PDEs.

2.1. Variable projection method for solving linear PDEs

Consider a domain $\Omega \subset \mathbf{R}^d$ ($d = 1$ to 3) and the following linear boundary-value problem on Ω,

$$Lu = f(\mathbf{x}), \tag{1a}$$

$$Bu = g(\mathbf{x}), \quad \text{on } \partial\Omega. \tag{1b}$$

In these equations $\mathbf{x} = (x_1, \ldots, x_d)$ denotes the coordinate, $u(\mathbf{x})$ is the field solution to be solved for, L denotes a linear differential operator, B denotes a linear algebraic or differential operator on the boundary $\partial\Omega$ representing the boundary conditions, and $f(\mathbf{x})$ and $g(\mathbf{x})$ are prescribed non-homogeneous terms in the domain or on the boundary. We assume that L may include linear differential operators with respect to the time t (e.g. $\frac{\partial}{\partial t}$, $\frac{\partial^2}{\partial t^2}$). In such a case, this becomes an initial boundary-value problem, and we treat the time t in the same way as the spatial coordinates. We designate the last coordinate x_d as t, and Ω becomes a spatial–temporal domain. Accordingly, we assume that the boundary condition (1b) in this case should include appropriate initial condition(s) with respect to t, which will be imposed only on the portion of $\partial\Omega$ corresponding to the initial condition(s). The point here is that the Eqs. (1a)–(1b)

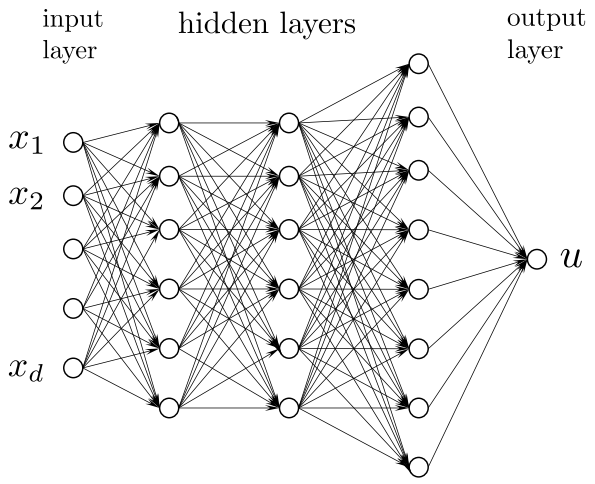

Fig. 1. Sketch of the neural network architecture (with 3 hidden layers).

may denote a time-dependent problem, and we will not distinguish the stationary and time-dependent cases in the following discussions. We assume that the problem (1) is well-posed.

We approximate the solution field $u(\mathbf{x})$ by a feed-forward neural network [2] with $(K+1)$ layers, where K is an integer satisfying $K \geqslant 2$; see Fig. 1 for a sktech of the network architecture with three hidden layers. The input layer (layer 0) of the neural network contains d nodes which represent the coordinate $\mathbf{x}$, and the output layer (layer K) contains 1 node which represents the solution u. The $(K-1)$ layers in between are the hidden layers. From layer to layer the network logic represents an affine transform followed by a node-wise function composition with an activation function $\sigma(\cdot)$ [2]. The coefficients of the affine transforms are referred to as the weight and bias coefficients of the neural network. For the convenience of presentation, we use the vector $[M_0, M_1, \ldots, M_K]$ to denote the architecture of the neural network, where M_i $(0 \leqslant i \leqslant K)$ denotes the number of nodes in layer i, with $M_0 = d$ and $M_K = 1$. We also use $M = M_{K-1}$ to denote the number of nodes in the last hidden layer in what follows. The weight/bias coefficients in all the hidden layers and in the output layer are the trainable parameters of the neural network.

In the current work, we make the assumption that the output layer contains no bias (or zero bias), and no activation function (or equivalently it uses the identity activation function $\sigma(x) = x$). So the output layer of the neural network is linear in this paper.

Let $\Phi_j(\boldsymbol{\theta}, \mathbf{x})$ $(1 \leqslant j \leqslant M)$ denote the output fields of the last hidden layer, where $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_{N_h})^T$ denotes the vector of weight/bias coefficients in all the hidden layers of the network, with $N_h = \sum_{i=1}^{K-1} M_i(M_{i-1}+1)$. Then we have the following expansion relation,

$$u(\mathbf{x}) = \sum_{j=1}^{M} \beta_j \Phi_j(\boldsymbol{\theta}, \mathbf{x}) = \boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})\boldsymbol{\beta} \tag{2}$$

where $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}) = [\Phi_1(\boldsymbol{\theta}, \mathbf{x}), \ldots, \Phi_M(\boldsymbol{\theta}, \mathbf{x})]$ denotes the set of output fields of the last hidden layer, and $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_M]^T$ is the vector of weight coefficients of the output layer. Note that $(\boldsymbol{\theta}, \boldsymbol{\beta})$ are the trainable parameters of the neural network. Note also that M represents the number of nodes in the last hidden layer, as well as the number of output-layer coefficients.

We choose a set of N $(N \geqslant 1)$ collocation points on Ω, which can be chosen according to a certain distribution (e.g. random, uniform). Among them N_b $(1 \leqslant N_b \leqslant N-1)$ collocation points reside on the boundary $\partial\Omega$, and the rest of the points are from the interior of Ω. We use $\mathbb{X}$ to denote the set of all the collocation points and $\mathbb{X}_b$ to denote the set of collocation points on $\partial\Omega$. In the current paper, for simplicity we assume that Ω is a rectangular domain, given by the interval $[a_i, b_i]$ $(1 \leqslant i \leqslant d)$ in the ith direction. We employ a uniform set of grid points (including the boundary end points) in each direction as the collocation points for the numerical tests in Section 3.

The input training data to the neural network consist of the coordinates of the all the N collocation points on Ω. We use the $N \times d$ matrix $\mathbf{X}$ to denote the input data. Each row of $\mathbf{X}$ denotes the coordinates of a collocation

point. Let the $N \times 1$ matrix $\mathbf{U}$ (column vector) denote the output data of the neural network, which represents the solution field $u(\mathbf{x})$ evaluated on all the N collocation points. We use the $N \times M$ matrix $\boldsymbol{\Psi}$ to denote the output data of the last hidden layer of the neural network, which represents the output fields $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ of the last hidden layer evaluated on all the N collocation points.

Inserting the expansion (2) into (1), and enforcing Eq. (1a) on all the collocation points from $\mathbb{X}$ and Eq. (1b) on all the boundary collocation points from $\mathbb{X}_b$, we arrive at the following system,

$$\sum_{j=1}^{M} \left[L\,\Phi_j(\boldsymbol{\theta}, \mathbf{x}_p) \right] \beta_j = f(\mathbf{x}_p), \quad 1 \leqslant p \leqslant N, \ \text{where } \mathbf{x}_p \in \mathbb{X}, \tag{3a}$$

$$\sum_{j=1}^{M} \left[B\,\Phi_j(\boldsymbol{\theta}, \mathbf{x}_q) \right] \beta_j = g(\mathbf{x}_q), \quad 1 \leqslant q \leqslant N_b, \ \text{where } \mathbf{x}_q \in \mathbb{X}_b. \tag{3b}$$

This is a system of $(N + N_b)$ algebraic equations about the trainable parameters $(\boldsymbol{\theta}, \boldsymbol{\beta})$, with $(N_h + M)$ unknowns. Note that for a given $\boldsymbol{\theta}$ the terms $L\,\Phi_j(\boldsymbol{\theta}, \mathbf{x}_p)$ and $B\,\Phi_j(\boldsymbol{\theta}, \mathbf{x}_q)$ in the above equations can be computed by forward evaluations of the neural network and auto-differentiations.

We seek a least squares solution for $(\boldsymbol{\theta}, \boldsymbol{\beta})$ to the system (3). This system is linear with respect to $\boldsymbol{\beta}$, and nonlinear with respect to $\boldsymbol{\theta}$. This leads to a separable nonlinear least squares problem. Therefore we adopt the variable projection approach [1] for the least squares solution of the system (3).

To make the formulation more compact, we re-write the system (3) into a matrix form,

$$\mathbf{H}(\boldsymbol{\theta})\boldsymbol{\beta} = \mathbf{S}, \quad \text{where} \quad \mathbf{H}(\boldsymbol{\theta}) = \begin{bmatrix} \vdots \\ L\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_p) \\ \vdots \\ \hdashline \vdots \\ B\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_q) \\ \vdots \end{bmatrix}_{(N+N_b)\times M}, \quad \mathbf{S} = \begin{bmatrix} \vdots \\ f(\mathbf{x}_p) \\ \vdots \\ \hdashline \vdots \\ g(\mathbf{x}_q) \\ \vdots \end{bmatrix}_{(N+N_b)\times 1}. \tag{4}$$

For any given $\boldsymbol{\theta}$, the least squares solution for the linear parameters $\boldsymbol{\beta}$ to this system is given by

$$\boldsymbol{\beta} = [\mathbf{H}(\boldsymbol{\theta})]^{+}\,\mathbf{S}, \tag{5}$$

where the superscript in $\mathbf{H}^{+}$ denotes the Moore–Penrose pseudo-inverse of $\mathbf{H}$. Define the residual function of the system (4) by

$$\mathbf{r}(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta})\boldsymbol{\beta} - \mathbf{S} = \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{S} - \mathbf{S}, \tag{6}$$

where the linear parameter $\boldsymbol{\beta}$ has been eliminated by using Eq. (5). We compute the optimal nonlinear parameters $\boldsymbol{\theta}_{opt}$ by minimizing the Euclidean norm of the residual function $\mathbf{r}$,

$$\boldsymbol{\theta}_{opt} = \arg\min_{\boldsymbol{\theta}} \frac{1}{2}\|\mathbf{r}(\boldsymbol{\theta})\|^2 = \arg\min_{\boldsymbol{\theta}} \frac{1}{2}\|\mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{S} - \mathbf{S}\|^2, \tag{7}$$

where $\|\cdot\|$ denotes the Euclidean norm. After $\boldsymbol{\theta}_{opt}$ is obtained, we can compute the optimal linear parameters $\boldsymbol{\beta}_{opt}$ based on Eq. (5) or by solving Eq. (4) using the linear least squares method. Outlined above is the essence of the variable projection approach for solving the system (3) for $(\boldsymbol{\theta}, \boldsymbol{\beta})$.

The problem represented by (7) is a nonlinear least squares problem about $\boldsymbol{\theta}$ only, where the linear parameter $\boldsymbol{\beta}$ has been eliminated. We solve this problem by a Gauss–Newton algorithm combined with a trust region strategy. Specifically, in the current paper we solve this problem by employing the nonlinear least squares library routine "scipy.optimize.least_squares" from the scipy package in Python, which implements the Gauss–Newton method together with a trust region reflective algorithm [72,73].

The scipy routine "least_squares()" requires two functions as input, which are needed by the Gauss–Newton algorithm. These are, for any given $\boldsymbol{\theta}$,

- a function for computing the residual $\mathbf{r}(\boldsymbol{\theta})$, and
- a function for computing the Jacobian matrix $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$.

Algorithm 1: Computing the residual $\mathbf{r}(\boldsymbol{\theta})$

input : $\boldsymbol{\theta}$; input data $\mathbf{X}$ to neural network; source data $\mathbf{S}$.
output: $\mathbf{r}(\boldsymbol{\theta})$.

1 update the hidden-layer coefficients of the neural network by $\boldsymbol{\theta}$
2 **if** $\boldsymbol{\theta} = \boldsymbol{\theta}_s$ **then**
3 retrieve $\mathbf{H}(\boldsymbol{\theta}_s)$, and set $\mathbf{H}(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta}_s)$
4 retrieve $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta}_s)$, and set $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta}) = \boldsymbol{\beta}^{LS}(\boldsymbol{\theta}_s)$
5 **else**
6 compute $\mathbf{H}(\boldsymbol{\theta})$ using the input data $\mathbf{X}$
7 solve equation (4) by the linear least squares method to get $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta})$
8 set $\boldsymbol{\theta}_s = \boldsymbol{\theta}$, and save $\mathbf{H}(\boldsymbol{\theta})$ and $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta})$
9 **end**
10 compute $\mathbf{r}(\boldsymbol{\theta})$ by equation (8)

The computation for $\mathbf{r}(\boldsymbol{\theta})$ is straightforward. For a given $\boldsymbol{\theta}$, we first solve Eq. (4) by the linear least squares method for the minimum-norm least squares solution $\boldsymbol{\beta}^{LS}$. Then we compute the residual according to Eq. (6) as follows,

$$\mathbf{r}(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta})\boldsymbol{\beta}^{LS} - \mathbf{S}. \tag{8}$$

Note that the Moore–Penrose inverse $\mathbf{H}^{+}(\boldsymbol{\theta})$ is not explicitly computed in the implementation. In the current paper we employ the linear least squares routine "scipy.linalg.lstsq" from scipy to solve (4) for $\boldsymbol{\beta}^{LS}$. The computation for $\mathbf{r}(\boldsymbol{\theta})$ is summarized in the Algorithm 1.

Remark 2.1. Let us elaborate on, for a given $\boldsymbol{\theta}$ and the input data $\mathbf{X}$, how to compute the matrix $\mathbf{H}(\boldsymbol{\theta})$ on line 6 of Algorithm 1. As defined in (4), $H(\boldsymbol{\theta})$ consists of the terms $L\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_p)$ ($\mathbf{x}_p \in \mathbb{X}$) and $B\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_q)$ ($\mathbf{x}_q \in \mathbb{X}_b$). These terms involve the output fields of the last hidden layer $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$, and their derivatives up to a certain order, evaluated on all the collocation points. All these terms can be computed by evaluating the neural network on the input data $\mathbf{X}$ and by auto-differentiations. Specifically, in our implementation we have created a sub-model to the neural network in Keras, with the neural network's input as its input and with the output of the neural network's last hidden layer as the sub-model's output. Let us refer to this sub-model as the last-hidden-layer-model. Let m denote the order of the PDE (1a), and we assume that the hidden-layer coefficients have been updated by the given $\boldsymbol{\theta}$. Then computing $\mathbf{H}(\boldsymbol{\theta})$ involves the following procedure:

(i) evaluate the last-hidden-layer-model on the input $\mathbf{X}$ to get $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ on all the collocation points;
(ii) compute the derivatives of $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ with respect to $\mathbf{x}$, up to the order m, on all the collocation points by a forward-mode auto-differentiation;
(iii) compute $L\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ on all the collocation points based on the data for $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ and its derivatives;
(iv) extract the boundary data (i.e. on the boundary collocation points) for $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ and its derivatives from those data attained from steps (i) and (ii);
(v) compute $B\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ based on the boundary data for $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ and its derivatives;
(vi) assemble $L\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ (on all the collocation points) and $B\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ (on the boundary collocation points) to form $\mathbf{H}(\boldsymbol{\theta})$ based on Eq. (4).

Note that we employ the forward-mode auto-differentiations to compute the derivatives of $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ in step (ii) above, because the number of nodes in the last hidden layer (M) is typically much larger than that in the input layer (d). In this case the forward-mode auto-differentiation is significantly faster than the reverse-mode auto-differentiation. In our implementation we have used the "ForwardAccumulator" from the Tensorflow library for the forward-mode auto-differentiations.

For computing the Jacobian matrix $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$ we consider the following formula, which is due to [1],

$$\begin{aligned}\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}} &= \left[\mathbf{I} - \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\right] \frac{\partial \mathbf{H}}{\partial \boldsymbol{\theta}} \mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{S} \;+\; \left[\mathbf{H}^{+}(\boldsymbol{\theta})\right]^{T} \frac{\partial \mathbf{H}^{T}}{\partial \boldsymbol{\theta}} \left[\mathbf{I} - \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\right] \mathbf{S} \\ &\approx \left[\mathbf{I} - \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\right] \frac{\partial \mathbf{H}}{\partial \boldsymbol{\theta}} \mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{S},\end{aligned} \tag{9}$$

where $\mathbf{I}$ denotes the identity matrix and Eq. (6) has been used. Note that here we have adopted the simplification suggested by [34] to keep only the first term for an approximation of $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$. So the Jacobian matrix is computed only approximately. This greatly simplifies the computation, and as observed in [28,34] only slightly or moderately increases the number of Gauss–Newton iterations.

In light of (9), we compute the approximate Jacobian matrix as follows. For any given $\boldsymbol{\theta}$, note that

$$\mathbf{J}_0(\boldsymbol{\theta}) \equiv \frac{\partial \mathbf{H}}{\partial \boldsymbol{\theta}} \mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{S} = \frac{\partial \mathbf{H}}{\partial \boldsymbol{\theta}} \boldsymbol{\beta}^{LS} = \frac{\partial \mathbf{V}}{\partial \boldsymbol{\theta}}, \tag{10}$$

where $\boldsymbol{\beta}^{LS}$ is the least squares solution of (4), and

$$\mathbf{V} = \mathbf{H}(\boldsymbol{\theta})\boldsymbol{\beta}_c^{LS}, \quad \text{with } \boldsymbol{\beta}_c^{LS} = \boldsymbol{\beta}^{LS}\big|_{\boldsymbol{\theta}}. \tag{11}$$

Here $\boldsymbol{\beta}_c^{LS}$ is a *constant* vector that equals $\boldsymbol{\beta}^{LS}$ at the given $\boldsymbol{\theta}$. The vector $\mathbf{V}(\boldsymbol{\theta})$ of length $(N + N_b)$ represents the field $\begin{bmatrix} Lu(\mathbf{x}) \\ Bu(\mathbf{x}) \end{bmatrix}$ evaluated on the collocation points (and the boundary collocation points), with $\boldsymbol{\theta}$ as the hidden-layer coefficients and $\boldsymbol{\beta}_c^{LS}$ as the output-layer coefficients in the neural network. We would like to emphasize that $\boldsymbol{\beta}_c^{LS}$ is considered to be constant and does not depend on $\boldsymbol{\theta}$ when computing $\mathbf{J}_0(\boldsymbol{\theta}) = \frac{\partial \mathbf{V}}{\partial \boldsymbol{\theta}}$. For a given $\boldsymbol{\theta}$, $\mathbf{J}_0(\boldsymbol{\theta})$ can be computed by an auto-differentiation of the neural network.

In light of (10) we transform (9) into

$$\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}} = \mathbf{J}_0(\boldsymbol{\theta}) - \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{J}_0(\boldsymbol{\theta}) = \mathbf{J}_0(\boldsymbol{\theta}) - \mathbf{J}_1(\boldsymbol{\theta}). \tag{12}$$

The term $\mathbf{J}_1(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta})\mathbf{H}^{+}(\boldsymbol{\theta})\mathbf{J}_0(\boldsymbol{\theta})$ can be computed as follows. For any given $\boldsymbol{\theta}$, we first solve the following system for the matrix $\mathbf{K}(\boldsymbol{\theta})$ by the linear least squares method,

$$\mathbf{H}(\boldsymbol{\theta})\mathbf{K}(\boldsymbol{\theta}) = \mathbf{J}_0(\boldsymbol{\theta}). \tag{13}$$

Then we compute $\mathbf{J}_1(\boldsymbol{\theta})$ by

$$\mathbf{J}_1(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta})\mathbf{K}(\boldsymbol{\theta}). \tag{14}$$

Therefore, in order to compute the Jacobian matrix we first solve equation (4) for $\boldsymbol{\beta}^{LS}$ by the linear least squares method, and then use (10) to compute $\mathbf{J}_0(\boldsymbol{\theta})$. We then compute $\mathbf{J}_1(\boldsymbol{\theta})$ by Eqs. (13) and (14). Finally the Jacobian matrix $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$ is computed by Eq. (12). These computations involve only the linear least squares method and the auto-differentiations of the neural network. The computation for the Jacobian matrix is summarized in Algorithm 2.

Remark 2.2. Let us elaborate on how to compute the matrix $\mathbf{J}_0(\boldsymbol{\theta})$, which has a dimension $(N + N_b) \times N_h$ (N_h denoting the total number of hidden-layer coefficients), on the lines 10 and 11 in Algorithm 2. This is for a given $\boldsymbol{\theta}$, $\boldsymbol{\beta}$ ($\boldsymbol{\beta} = \boldsymbol{\beta}_c^{LS}$), and the input data $\mathbf{X}$ to the neural network. Based on Eq. (11), the column vector $\mathbf{V}(\boldsymbol{\theta})$ consists of the terms $Lu(\mathbf{x}_p)$ ($\mathbf{x}_p \in \mathbb{X}$) and $Bu(\mathbf{x}_q)$ ($\mathbf{x}_q \in \mathbb{X}_b$), where $u(\mathbf{x})$ is the output field of the neural network obtained with the given $(\boldsymbol{\theta}, \boldsymbol{\beta}_c^{LS})$ as the hidden-layer coefficients and the output-layer coefficients, respectively. It should be noted that Lu and Bu involve the derivatives of $u(\mathbf{x})$ with respect to $\mathbf{x}$ (not $\boldsymbol{\theta}$). Based on Eq. (10), the matrix $\mathbf{J}_0(\boldsymbol{\theta})$ consists of the terms $\frac{\partial (Lu)}{\partial \boldsymbol{\theta}}\Big|_{(\boldsymbol{\theta}, \mathbf{x}_p)}$ ($\mathbf{x}_p \in \mathbb{X}$) and $\frac{\partial (Bu)}{\partial \boldsymbol{\theta}}\Big|_{(\boldsymbol{\theta}, \mathbf{x}_q)}$ ($\mathbf{x}_q \in \mathbb{X}_b$). These terms can be computed by evaluating the neural network on the input data $\mathbf{X}$ and by auto-differentiations with respect to $\mathbf{x}$ and $\boldsymbol{\theta}$. We assume again that the PDE (1a) is of the mth order. Given $(\boldsymbol{\theta}, \boldsymbol{\beta}, \mathbf{X})$, we compute $\mathbf{J}_0(\boldsymbol{\theta})$ specifically by the following procedure:

(i) update the hidden-layer coefficients of the neural network by $\boldsymbol{\theta}$, and update the output-layer coefficients by $\boldsymbol{\beta}$;

(ii) evaluate the neural network on the input $\mathbf{X}$ to obtain the output field $u(\mathbf{x})$ on all the collocation points;

Algorithm 2: Computing the Jacobian matrix $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$

input : $\boldsymbol{\theta}$; input data $\mathbf{X}$ to neural network; source data $\mathbf{S}$.
output: $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$.

1 update the hidden-layer coefficients of the neural network by $\boldsymbol{\theta}$
2 **if** $\boldsymbol{\theta} = \boldsymbol{\theta}_s$ **then**
3 retrieve $\mathbf{H}(\boldsymbol{\theta}_s)$, and set $\mathbf{H}(\boldsymbol{\theta}) = \mathbf{H}(\boldsymbol{\theta}_s)$
4 retrieve $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta}_s)$, and set $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta}) = \boldsymbol{\beta}^{LS}(\boldsymbol{\theta}_s)$
5 **else**
6 compute $\mathbf{H}(\boldsymbol{\theta})$ using the input data $\mathbf{X}$
7 solve equation (4) by the linear least squares method to get $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta})$
8 set $\boldsymbol{\theta}_s = \boldsymbol{\theta}$, and save $\mathbf{H}(\boldsymbol{\theta})$ and $\boldsymbol{\beta}^{LS}(\boldsymbol{\theta})$
9 **end**
10 compute $\mathbf{V}(\boldsymbol{\theta})$ by equation (11)
11 compute $\mathbf{J}_0(\boldsymbol{\theta})$ based on equation (10) by auto-differentiations
12 solve equation (13) for $\mathbf{K}(\boldsymbol{\theta})$ by the linear least squares method
13 compute $\mathbf{J}_1(\boldsymbol{\theta})$ by equation (14)
14 compute $\frac{\partial \mathbf{r}}{\partial \boldsymbol{\theta}}$ by equation (12)

(iii) compute the derivatives of $u(\mathbf{x})$ with respect to $\mathbf{x}$, up to the order m, by a reverse-mode auto-differentiation;
(iv) compute the derivative, with respect to the hidden-layer coefficients, for $u(\mathbf{x})$ and for its derivatives with respect to $\mathbf{x}$ from steps (ii) and (iii), on all the collocation points by a reverse-mode auto-differentiation;
(v) compute $\frac{\partial (Lu)}{\partial \boldsymbol{\theta}}$ on all the collocation points based on the data for $u(\mathbf{x})$ and its derivatives from the previous step;
(vi) extract the boundary data (i.e. on the boundary collocation points) for $u(\mathbf{x})$ and its derivatives from the data obtained from step (iv);
(vii) compute $\frac{\partial (Bu)}{\partial \boldsymbol{\theta}}$ based on the boundary data for $u(\mathbf{x})$ and its derivatives from the previous step;
(viii) assemble $\frac{\partial (Lu)}{\partial \boldsymbol{\theta}}$ (on all the collocation points) and $\frac{\partial (Bu)}{\partial \boldsymbol{\theta}}$ (on the boundary collocation points) to form $\mathbf{J}_0(\boldsymbol{\theta})$.

When computing the derivatives of $u(\mathbf{x})$ with respect to $\mathbf{x}$ and with respect to the hidden-layer coefficients in the steps (iii) and (iv) above, in our implementation we have employed a vectorized map (tf.vectorized_map) together with the gradient tape (tf.GradientTape) in the Tensorflow library to vectorize the gradient computations.

Remark 2.3. In Algorithms 1 and 2, we have saved the matrix $\mathbf{H}(\boldsymbol{\theta})$ and the vector $\boldsymbol{\beta}^{LS}$ when they are computed for a new $\boldsymbol{\theta}$; see the lines 2 to 9 in both algorithms. The goal of this extra storage is to save computations. During the Gauss–Newton iterations, the Algorithm 2 is typically invoked to compute the Jacobian matrix for the same $\boldsymbol{\theta}$, following the call to the Algorithm 1 for computing the residual $\mathbf{r}(\boldsymbol{\theta})$. In this case one avoids the re-computation of the matrix $\mathbf{H}(\boldsymbol{\theta})$ and the vector $\boldsymbol{\beta}^{LS}$ for the same $\boldsymbol{\theta}$.

To solve the system (3a)–(3b) with the variable projection approach, we first solve the reduced problem (7) for $\boldsymbol{\theta}$ by the nonlinear least squares method, and then we solve Eq. (4) for $\boldsymbol{\beta}$ by the linear least squares method. To make the nonlinear least squares computation for (7) more robust (from being trapped to local minima), in our implementation we have incorporated a perturbation to the initial guess and a sub-iteration procedure, in a way analogous to the NLLSQ-perturb method from [21]. The sub-iteration procedure will be triggered if the nonlinear least squares computation fails to converge or the converged cost value is not small enough. The overall variable projection algorithm with perturbations for solving the system (3) is summarized in Algorithm 3. The perturbations to the initial guess of the nonlinear least squares computation are generated on the lines 6 to 13 in Algorithm 3.

Remark 2.4. In Algorithm 3, when generating the perturbation magnitude δ_1, we have incorporated a preferred perturbation magnitude δ_{pref} and a preference probability p. Here δ_{pref} keeps the last perturbation magnitude δ_1 that

Algorithm 3: Variable projection algorithm with perturbations

input : input data $\mathbf{X}$ to neural network; source data $\mathbf{S}$; initial guess $\boldsymbol{\theta}_0$; maximum perturbation magnitude $\delta > 0$; preference probability p ($p \in [0, 1]$), with default value $p = 0.5$.
output: $\boldsymbol{\theta}$ and $\boldsymbol{\beta}$.

```
1  call scipy.optimize.least_squares routine to solve (7), using θ_0 as the initial guess, with Algorithms 1 and 2
     as input arguments
2  set θ ← returned solution, and c ← returned cost
3  if c is above a threshold then
4      set δ_pref = None
5      for i ← 1 to maximum-number-of-sub-iterations do
6          generate a uniform random number ξ ∈ [0, 1]
7          if (δ_pref is not None) and (ξ < p) then
8              generate a uniform random number δ_1 ∈ [0, min(1.1δ_pref, δ)]
9          else
10             generate a uniform random number δ_1 ∈ [0, δ]
11         end
12         generate a uniform random vector Δθ of the same shape as θ on the interval [−δ_1, δ_1]
13         set ϑ_0 ← θ + Δθ
14         call scipy.optimize.least_squares routine to solve (7), using ϑ_0 as the initial guess, with
             Algorithms 1 and 2 as input arguments
15         if the returned cost is less than c then
16             set θ ← returned solution, and c ← returned cost
17             set δ_pref = δ_1
18         end
19         if c is not above a threshold then
20             break
21         end
22     end
23 end
24 solve equation (4) for β by the linear least squares method
```

has resulted in a reduction in the converged cost. The lines 6 to 11 of Algorithm 3 basically means that, with a probability p, we will generate the next perturbation magnitude δ_1 based on the preferred magnitude δ_{pref}. Otherwise, we will generate the next perturbation magnitude based on the original maximum magnitude δ. After the algorithm hits upon a favorable perturbation magnitude, the employment of δ_{pref} and the probability p tends to promote the use of this value. When the algorithm is close to convergence, this also tends to reduce the amount of the perturbation, which is conducive to achieving convergence. In the current paper we employ a preference probability $p = 0.5$ with the variable projection algorithm for all the numerical tests in Section 3.

Remark 2.5. If the problem consisting of Eqs. (1a)–(1b) is time dependent, for longer-time or long-time simulations we employ the block time marching scheme from [21] together with the variable projection algorithm developed here. The basic idea is as follows. If the domain Ω has a large dimension in time, we first divide the temporal dimension into a number of windows (referred to as time blocks), so that each time block has a moderate size in time. We solve the problem using the variable projection algorithm on the spatial–temporal domain of each time block individually and successively. After one time block is computed, the field solution (and also possibly its derivatives) evaluated at the last time instant of this block is used as the initial condition(s) for the subsequent time block. We refer the reader to [21] for more detailed discussions of the block time marching scheme.

Remark 2.6. We next comment on the VarPro algorithm combined with domain decomposition and local neural networks for solving the system (1). In the above discussions we have represented the solution field to the boundary value problem (1) by a single feed-forward neural network over the entire domain Ω. This can be considered as a "global" method. Alternatively, as discussed in [21], one can decompose the domain Ω into sub-domains and represent the solution on each sub-domain by a local feed-forward neural network, and then impose C^k (with an appropriate k) continuity conditions across the sub-domain boundaries. We can combine VarPro and this local function representation based on domain decomposition and local neural networks for solving PDEs. Note that in [21] the hidden-layer coefficients of the local neural networks are fixed random values and are not trainable, and only the output-layer coefficients are trainable parameters. In the current case, unlike in [21], the trainable parameters consist of the coefficients in both the hidden layers and the output layers of the local neural networks. Note also that all the local neural networks are coupled due to the C^k continuity conditions. On each sub-domain we choose a set of collocation points, with a subset of these points residing on the sub-domain boundaries. To solve the system (1), we enforce Eq. (1a) on all the collocation points from each sub-domain, and enforce Eq. (1b) on those collocation points of each sub-domain that reside on the domain boundary $\partial\Omega$. In addition, we enforce the C^k continuity conditions of the solution field on those collocation points of each sub-domain that reside on the common sub-domain boundaries of adjacent sub-domains. We refer the reader to [21] for detailed discussions of the enforcement of these equations/conditions. These operations result in a system of equations about the hidden-/output-layer coefficients of the local neural networks. We seek a least squares solution to this system, leading to a separable nonlinear least squares problem. The linear parameters of this problem consist of the set of output-layer coefficients of all the local neural networks, and the nonlinear parameters consist of the set of hidden-layer coefficients of all the local neural networks. The variable projection idea can be used to solve this problem. We reformulate this problem by VarPro to eliminate the linear parameters and arrive at a reduced problem about the nonlinear parameters only. Then the method as discussed in this section can be employed to determine the nonlinear parameters first, which in turn are used to compute the linear parameters. The procedure outlined above can be considered as a "local" version of the VarPro method. The local VarPro method would be more favorable for PDE problems in which certain complex features (e.g. sharp gradient) may exist locally in the domain. In such a case one can exploit domain decomposition and the local VarPro method to better capture the local complex features of the solution.

Remark 2.7. It would be interesting to compare the current VarPro method with the extreme learning machine (ELM) method from [21,25] for solving PDEs. With ELM, the weight/bias coefficients in all the hidden layers of the neural network are pre-set to random values and are fixed, while the output-layer coefficients are computed by the linear least squares method for solving linear PDEs and by the nonlinear least squares method for solving nonlinear PDEs [21]. In [21,25] the hidden-layer coefficients are set and fixed to uniform random values generated on the interval $[-R_m, R_m]$, where R_m is a user-provided constant (hyperparameter). The constant R_m has an influence on the accuracy of ELM, and the optimal R_m value (denoted by R_{m0} in [25]) can be computed by the method from [25] based on the differential evolution algorithm. It is crucial to note that in ELM all the hidden-layer coefficients are fixed (not trained) once they are set.

With the VarPro method, the output-layer coefficients are always computed by the linear least squares method, once the hidden-layer coefficients are determined. The weight/bias coefficients in the hidden layers are determined by considering the reduced problem, which eliminates the linear output-layer coefficients. The hidden-layer coefficients are computed by solving the reduced problem using the nonlinear least squares method. With the VarPro approach, the hidden-layer coefficients of the neural network are trained/computed first by solving the reduced problem, and then the output-layer coefficients are computed by the linear least squares method afterwards. If the maximum number of iterations is set to zero in the nonlinear least squares solution of the reduced problem, the VarPro algorithm will be reduced to essentially the ELM method.

With the same neural network architecture and under the same settings, the VarPro method is in general significantly more accurate than the ELM method. In particular, VarPro can produce highly accurate solutions when the number of nodes in the last hidden layer is not large. In contrast, the result produced by ELM in this case is usually much less accurate or utterly inaccurate. VarPro achieves the higher accuracy at the price of the computational cost. Because VarPro needs to solve the reduced problem for the hidden-layer coefficients by a nonlinear least squares computation, its computational cost is usually much higher than that of the ELM method,

which only computes the output-layer coefficients by the linear least squares method (for linear PDEs). In numerical simulations with VarPro, we initialize the hidden-layer coefficients (i.e. the initial guess $\boldsymbol{\theta}_0$ in Algorithm 3) to uniform random values generated on the interval $[-R_m, R_m]$, with $R_m = 1$ in general (or with R_m set to a user-provided value). We observe from numerical experiments that the VarPro method is less sensitive or insensitive to the random coefficient initializations (the R_m constant) than ELM. We provide numerical experiments in Section 3 for comparisons between the VarPro and the ELM methods.

2.2. Newton-variable projection method for solving nonlinear PDEs

We next develop a method based on variable projection for solving nonlinear PDEs. The notations and settings here follow those of Section 2.1.

Consider the following nonlinear boundary value problem on the domain Ω in d dimensions,

$$Lu + F(u) = f(\mathbf{x}), \tag{15a}$$

$$Bu + G(u) = g(\mathbf{x}), \quad \text{on } \partial\Omega, \tag{15b}$$

where $F(u)$ and $G(u)$ are nonlinear operators on the solution field $u(\mathbf{x})$ and also possibly on its derivatives, and L, B, f and g have the same meanings as in the Eqs. (1a)–(1b). We assume that the highest-order term occurs in the linear differential operator L, and that the nonlinear terms $F(u)$ and $G(u)$ involve only the lower-order derivatives (if any). We again assume that the L operator may involve time derivatives. In such a case we treat the time t in the same way as the spatial coordinate $\mathbf{x}$, as discussed in Section 2.1. We assume that this problem is well-posed.

We approximate the field solution $u(\mathbf{x})$ to the system (15) by a feed-forward neural network with $(K+1)$ layers, following the same configurations and settings as discussed in Section 2.1. Substituting the expansion relation (2) for $u(\mathbf{x})$ into Eqs. (15a) and (15b), and enforcing these two equations on all the collocation points from $\mathbb{X}$ and on all the boundary collocation points from $\mathbb{X}_b$ respectively, we arrive at an algebraic system of $(N + N_b)$ equations about the $(N_h + M)$ unknown neural-network coefficients $(\boldsymbol{\theta}, \boldsymbol{\beta})$. We seek a least squares solution to this system, thus leading to a nonlinear least squares problem. This algebraic system, however, is nonlinear with respect to both $\boldsymbol{\theta}$ and $\boldsymbol{\beta}$, because of the nonlinear terms $F(u)$ and $G(u)$ in (15a)–(15b). This is not a separable nonlinear least squares problem. The variable projection approach apparently cannot be used for solving this system, at least with the above straightforward formulation.

To circumvent the above issue and enable the use of the variable projection strategy, we consider the linearization of the system (15a)–(15b) with the Newton's method. Let u^k denote the approximation of the solution at the kth Newton iteration. We linearize this system as follows,

$$Lu^{k+1} + F(u^k) + F'(u^k)\left(u^{k+1} - u^k\right) = f(\mathbf{x}), \tag{16a}$$

$$Bu^{k+1} + G(u^k) + G'(u^k)\left(u^{k+1} - u^k\right) = g(\mathbf{x}), \quad \text{on } \partial\Omega, \tag{16b}$$

where $F'(u)$ and $G'(u)$ denote the derivatives with respect to u. We further re-write the linearized system into,

$$Lu^{k+1} + F'(u^k)u^{k+1} = f(\mathbf{x}) - F(u^k) + F'(u^k)u^k, \tag{17a}$$

$$Bu^{k+1} + G'(u^k)u^{k+1} = g(\mathbf{x}) - G(u^k) + G'(u^k)u^k, \quad \text{on } \partial\Omega. \tag{17b}$$

Given u^k, this system represents a linear boundary value problem about the updated approximation field u^{k+1}. Therefore, the VarPro/ANN algorithm developed in Section 2.1 can be used to solve this linearized system (17a)–(17b) for u^{k+1}. Upon convergence of the Newton iteration, the solution to the original nonlinear system (15a)–(15b) will be obtained and represented by the neural-network coefficients.

Remark 2.8. It is important to notice that the above formulation leads to a linearized system about the updated approximation field u^{k+1} directly. This linearization form is crucial to the high accuracy for solving nonlinear PDEs with the variable projection approach and artificial neural networks.

An alternative and perhaps more commonly-used form of linearization for the Newton's method is often formulated in terms of the increment field. Let

$$u^{k+1} = u^k + v, \tag{18}$$

where v is the increment field at the step k. Then the increment is given by the following linearized system,

$$Lv + F'(u^k)v = f(\mathbf{x}) - \left[Lu^k + F(u^k)\right], \tag{19a}$$

$$Bv + G'(u^k)v = g(\mathbf{x}) - \left[Bu^k + G(u^k)\right], \quad \text{on } \partial\Omega. \tag{19b}$$

So the increment field v can be computed by the variable projection approach from the above system. The updated approximation u^{k+1} is given by Eq. (18).

There are two issues with the form of linearization given by (19) when using variable projection and artificial neural networks. First, with this form u^{k+1} can only be computed in the physical space (i.e. on the collocation points), and it is not represented in terms of the neural network (i.e. given by the network coefficients). Note that with the system (19) the increment field v is computed by the VarPro/ANN algorithm and is represented by the hidden-layer and output-layer coefficients of the neural network. But u^{k+1} is computed by Eq. (18). This can only be performed in the physical space, not in terms of the neural-network coefficients, due to the nonlinearity of the network output with respect to the hidden-layer coefficients. Second, upon convergence of the Newton iteration, the solution to the nonlinear system (i.e. the converged u^{k+1}) is given in the physical space (on the collocation points), not represented by the neural network. Therefore, one needs to additionally convert this solution from physical space to the neural network representation, by solving a function approximation problem using the neural network and variable projection. This extra step is necessary in order to evaluate the solution field on the points other than the training collocation points in the domain.

The form of linearization given by (16), on the other hand, does not suffer from these issues. The updated approximation field u^{k+1} computed by the variable projection method is directly represented by the neural-network coefficients, as well as the solution to the original nonlinear system upon convergence. We observe that the solution obtained based on the formulation (16) is considerably more accurate, typically by two orders of magnitude or more, than that obtained based on the formulation (19). It should be noted that the system (19) can be transformed into the system (16) by the substitution $v = u^{k+1} - u^k$.

Within each Newton iteration we solve the linear boundary value problem (17) using the variable projection method. In order to make the following discussions more concise, we introduce the following notation to drop the superscripts,

$$\begin{cases} u(\mathbf{x}) = u^{k+1}(\mathbf{x}), \quad w(\mathbf{x}) = u^k(\mathbf{x}), \\ f_a(\mathbf{x}) = f(\mathbf{x}) - F(u^k) + F'(u^k)u^k, \quad g_a(\mathbf{x}) = g(\mathbf{x}) - G(u^k) + G'(u^k)u^k. \end{cases} \tag{20}$$

Then the system (17) is re-written into,

$$Lu + F'(w)u = f_a(\mathbf{x}), \tag{21a}$$

$$Bu + G'(w)u = g_a(\mathbf{x}), \quad \text{on } \partial\Omega. \tag{21b}$$

Let us next consider the solution of (21) with the variable projection approach. This system is similar to (1). The solution procedure mirrors that of Section 2.1. So we only summarize the most important steps below. We use a feed-forward neural network to represent the solution $u(\mathbf{x})$ to the system (21), with the same settings and configurations for the neural network and the collocation points as given in Section 2.1. Substituting the expansion (2) into (21), and enforcing Eq. (21a) on all the collocation points and Eq. (21b) on all the boundary collocation points, we get the following system in matrix form,

$$\mathbf{H}(\boldsymbol{\theta})\boldsymbol{\beta} = \mathbf{S}, \text{ where } \mathbf{H}(\boldsymbol{\theta}) = \begin{bmatrix} \vdots \\ L\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_p) + F'(w)\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_p) \\ \vdots \\ \hdashline \vdots \\ B\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_q) + G'(w)\,\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_q) \\ \vdots \end{bmatrix}_{(N+N_b)\times M}, \quad \mathbf{S} = \begin{bmatrix} \vdots \\ f_a(\mathbf{x}_p) \\ \vdots \\ \hdashline \vdots \\ g_a(\mathbf{x}_q) \\ \vdots \end{bmatrix}_{(N+N_b)\times 1}, \tag{22}$$

where $\mathbf{x}_p \in \mathbb{X}$ and $\mathbf{x}_q \in \mathbb{X}_b$. Following the same developments as given by Eqs. (5) and (7), we arrive at the reduced nonlinear least squares problem (7) about $\boldsymbol{\theta}$, with the understanding that the terms $\mathbf{H}(\boldsymbol{\theta})$ and $\mathbf{S}$ in all those equations

Algorithm 4: Newton-variable projection algorithm for the nonlinear problem (15).

input : input data $\mathbf{X}$ to neural network; source data $f(\mathbf{x}_p)$ ($\mathbf{x}_p \in \mathbb{X}$) and $g(\mathbf{x}_q)$ ($\mathbf{x}_q \in \mathbb{X}_b$); initial guess $u^0(\mathbf{x})$.

output: solution $u(\mathbf{x})$, represented by the coefficients of the neural network.

for $k \leftarrow 0$ **to** *maximum-number-of-newton-iterations* **do**
  compute the vector $\mathbf{R}$ by (23)
  if $\|\mathbf{R}\|$ *is below a tolerance* **then**
    break
  end
  compute $\mathbf{S}$ by the equations (22) and (20)
  call Algorithm 3, with "equation (4)" on line 24 therein replaced by "equation (22)", to obtain $(\boldsymbol{\theta}, \boldsymbol{\beta})$, which are the neural-network representation of u^{k+1} in the system (17)
  compute the vector $\Delta\mathbf{U}$ by (23)
  if $\|\Delta\mathbf{U}\|$ *is below a tolerance* **then**
    break
  end
end

are now defined by (22). We then invoke the Algorithm 3 to compute $(\boldsymbol{\theta}, \boldsymbol{\beta})$, with the understanding that on line 24 of that algorithm the "Eq. (4)" is now replaced by Eq. (22) when computing $\boldsymbol{\beta}$.

Remark 2.9. It should be noted that, depending on the form of the nonlinear operator $F(u)$, the terms $F'(w)\boldsymbol{\Phi}$ in the matrix $\mathbf{H}(\boldsymbol{\theta})$ may involve the derivatives of $\boldsymbol{\Phi}$. For example, with $F(u) = u\frac{\partial u}{\partial x}$, we have $F'(w)\boldsymbol{\Phi} = \frac{\partial w}{\partial x}\boldsymbol{\Phi} + w\frac{\partial \boldsymbol{\Phi}}{\partial x}$. The extra terms $F'(w)\boldsymbol{\Phi}$ and $G'(w)\boldsymbol{\Phi}$ in $\mathbf{H}(\boldsymbol{\theta})$ do not add to the difficulty in computing the matrices $\mathbf{H}(\boldsymbol{\theta})$ and $\mathbf{J}_0(\boldsymbol{\theta})$. Computing $\mathbf{H}(\boldsymbol{\theta})$ and $\mathbf{J}_0(\boldsymbol{\theta})$ follows the same procedures as outlined in Remarks 2.1 and 2.2. The only difference lies in that in $\mathbf{H}(\boldsymbol{\theta})$ one needs to additionally compute the $F'(w)\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_p)$ ($\mathbf{x}_p \in \mathbb{X}$) and $G'(w)\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x}_q)$ ($\mathbf{x}_q \in \mathbb{X}_b$) based on the data for $\boldsymbol{\Phi}(\boldsymbol{\theta}, \mathbf{x})$ and its derivatives on the collocation points. In $\mathbf{J}_0(\boldsymbol{\theta})$ one needs to additionally compute the $F'(w)\frac{\partial u}{\partial \boldsymbol{\theta}}\big|_{(\boldsymbol{\theta}, \mathbf{x}_p)}$ ($\mathbf{x}_p \in \mathbb{X}$) and $G'(w)\frac{\partial u}{\partial \boldsymbol{\theta}}\big|_{(\boldsymbol{\theta}, \mathbf{x}_q)}$ ($\mathbf{x}_q \in \mathbb{X}_b$) based on the data for $u(\mathbf{x})$ and its derivatives with respect to $\mathbf{x}$ and $\boldsymbol{\theta}$ on the collocation points.

To solve the nonlinear boundary value problem (15), we employ an overall Newton iteration. Within each iteration we invoke the variable projection method as given by Algorithm 3 to solve the system (17) for u^{k+1}, and the computed u^{k+1} is represented by the weight/bias coefficients of the artificial neural network. Upon convergence of the Newton iteration, the solution to the original nonlinear system (15) is given by the neural network, represented by the neural-network coefficients. In our implementation, we have considered two stopping criteria for the Newton iterations, based on the Euclidean norms of the residual vector $\mathbf{R}$ and the increment vector $\Delta\mathbf{U}$ defined by

$$\mathbf{R} = \begin{bmatrix} \vdots \\ f(\mathbf{x}_p) - Lu^k(\mathbf{x}_p) - F(u^k(\mathbf{x}_p)) \\ \vdots \\ \hdashline \vdots \\ g(\mathbf{x}_q) - Bu^k(\mathbf{x}_q) - G(u^k(\mathbf{x}_q)) \\ \vdots \end{bmatrix}_{(N+N_b)\times 1}, \qquad \Delta\mathbf{U} = \begin{bmatrix} \vdots \\ u^{k+1}(\mathbf{x}_p) - u^k(\mathbf{x}_p) \\ \vdots \end{bmatrix}_{N\times 1}. \tag{23}$$

The overall Newton-VarPro method with ANNs for solving the nonlinear system (15) is summarized in the Algorithm 4.

Remark 2.10. When solving the nonlinear problem (15) using the Newton-VarPro method (Algorithm 4), one can often turn off the initial-guess perturbations and sub-iterations in Algorithm 3 when invoking this algorithm to solve the system (17). This can be achieved by simply setting the "maximum-number-of-sub-iterations" to zero on line 5 of the Algorithm 3. In this case, if the converged cost from Algorithm 3 is not very small (above some threshold), this means that the returned u^{k+1} solution from that Newton step may not be that accurate. This inaccuracy, however, can be offset by the subsequent Newton iterations.

Remark 2.11. When solving nonlinear PDEs using the Newton-VarPro method, if the resolution is low (e.g. using a small number of training collocation points), we observe from numerical experiments that the Newton iteration may have difficulty reaching convergence within a specified maximum number of iterations. In such a case, increasing the resolution (e.g. increasing the number of collocation points) can typically improve the convergence of the Newton iteration. In the numerical experiments of Section 3 we typically employ a relative tolerance $1E-8$ for the Newton iterations (see the lines 3 and 9 of Algorithm 4).

3. Numerical examples

We use several numerical examples involving linear and nonlinear PDEs to illustrate the performance characteristics of the VarPro method. These problems are in two spatial dimensions or in one spatial dimension plus time. We also compare the simulation results of the current VarPro method and the ELM method from [21,25] to demonstrate the superior accuracy of the current method.

As stated previously, the current VarPro method is implemented in Python, using the Tensorflow (www.tensorflow.org) and Keras (keras.io) libraries. For the linear least squares method we employ the scipy routine "scipy.linalg.lstsq" in our implementation, which invokes the corresponding routine from the LAPACK library. For the nonlinear least squares method, we employ the scipy routine "scipy.optimize.least_squares" in our application code, which implements the Gauss–Newton method together with a trust region algorithm [72]. The differential operators acting on the output fields of the last hidden layer are computed by a forward-mode auto-differentiation in our implementation, as discussed in Remark 2.1. The data for the Jacobian matrix are computed by a reverse-mode auto-differentiation using a vectorized map together with the GradientTape in Tensorflow, as discussed in Remark 2.2. We would like to mention that in our implementation of the neural network, between the input layer and the first hidden layer, we have incorporated a lambda layer from Keras to normalize the input data $\mathbf{X}$ from the rectangular domain $\Omega = [a_1, b_1] \times \cdots \times [a_d, b_d]$ to the standard domain $[-1, 1]^d$.

As in our previous works [21,22,25], we employ a fixed seed value for the random number generators in the numerical experiments in each subsection, so that the reported results here can be exactly reproducible. We use the same seed for the random number generators from the Tensorflow library and from the numpy package. These seed values are 1 in Sections 3.1.1 and 3.2.1, 10 in Sections 3.1.2 and 3.2.2, and 22 in Section 3.2.3.

3.1. Linear examples

3.1.1. Poisson equation

We first consider the canonical two-dimensional (2D) Poisson equation on a unit square domain, $(x, y) \in [0, 1] \times [0, 1]$,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \tag{24a}$$

$$u(x, 0) = g_1(x), \quad u(x, 1) = g_2(x), \quad u(0, y) = g_3(y), \quad u(1, y) = g_4(y), \tag{24b}$$

where $u(x, y)$ is the field function to be solved for, $f(x, y)$ is a prescribed source term, and g_i $(1 \leqslant i \leqslant 4)$ denote the boundary data. We employ the following analytic solution to this problem in the tests,

$$u = \left[2\cos\left(\frac{3}{2}\pi x + \frac{2}{5}\pi\right) + \frac{3}{2}\cos\left(3\pi x - \frac{\pi}{5}\right) + \frac{1}{1+x^2}\right]\left[2\cos\left(\frac{3}{2}\pi y + \frac{2}{5}\pi\right) + \frac{3}{2}\cos\left(3\pi y - \frac{\pi}{5}\right) + \frac{1}{1+y^2}\right], \tag{25}$$

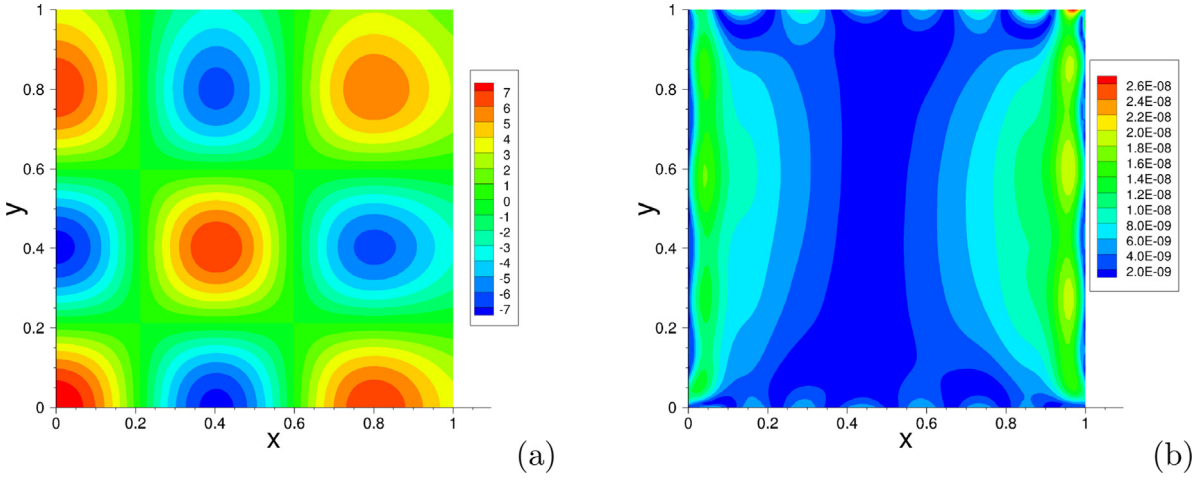

Fig. 2. Poisson equation: (a) Distribution of the exact solution. (b) Distribution of the absolute error of the VarPro solution. In (b), neural network [2, 125, 1], "cos" activation function, $Q = 15 \times 15$ uniform training collocation points.

Table 1
Poisson equation: Main simulation parameters of the VarPro method.

Parameter	Value	Parameter	Value
Neural network	$[2, M, 1]$ or $[2, 20, M, 1]$	Training points Q	$Q_1 \times Q_1$
M	Varied	Q_1	Varied
Activation function	cos, Gaussian	Testing points	$Q_2 \times Q_2$
Random seed	1	Q_2	101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	R_m	1.0
δ (Algorithm 3)	0.5, 1.0, 2.0, 5.0, or 7.0	p (Algorithm 3)	0.5
max-subiterations	5	Threshold (Algorithm 3)	$1E{-}12$

by choosing the source term f and the boundary data g_i $(1 \leqslant i \leqslant 4)$ appropriately. Fig. 2(a) shows the distribution of this analytic solution in the xy plane.

We employ feed-forward neural networks with one or two hidden layers, with the architecture given by $[2, M, 1]$ or $[2, 20, M, 1]$, where M is varied systematically or fixed at $M = 100$, 125 or 200. The two input nodes represent the coordinates (x, y), and the single output node represents the solution field $u(x, y)$. The activation function for the hidden nodes is either the cosine function, $\sigma(x) = \cos(x)$, or the Gaussian function, $\sigma(x) = e^{-x^2}$. The output layer is required to be linear (no activation function) and contain no bias.

We employ a uniform set of $Q = Q_1 \times Q_1$ grid points on the domain as the training collocation points, where Q_1 denotes the number of uniform grid points in each direction (including the two end points) and is varied systematically between around 5 and 30 in the tests. After the neural network is trained by the VarPro method on the $Q_1 \times Q_1$ collocation points, the neural network is evaluated on a much larger uniform set of $Q_2 \times Q_2$ grid points, where $Q_2 = 101$ for this problem, to obtain the solution $u(x, y)$. This solution is compared with the analytic solution (25) to compute the maximum (L^∞) and the root-mean-squares (rms, or L^2) errors. These maximum/rms errors are then recorded and referred to as the errors associated with the given neural network architecture and the training collocation points $Q = Q_1 \times Q_1$ for the VarPro method.

The main simulation parameters of the VarPro method are summarized in Table 1. The last three rows of this table pertain to the parameters in Algorithm 3. $\boldsymbol{\theta}_0$ denotes the initial guess to the hidden-layer coefficients in Algorithm 3, which are set to uniform random values generated on $[-R_m, R_m]$ with $R_m = 1.0$. When comparing the VarPro method and the ELM method, we also employ a value $R_m = R_{m0}$ with VarPro, where R_{m0} is the optimal R_m value corresponding to ELM computed using the method from [25]. δ and p in this table are the maximum perturbation magnitude and the preference probability in Algorithm 3, respectively. The "max-subiterations" here refers to the maximum-number-of-sub-iterations in Algorithm 3. The "threshold" here refers to the threshold on the lines 3 and 19 in Algorithm 3.

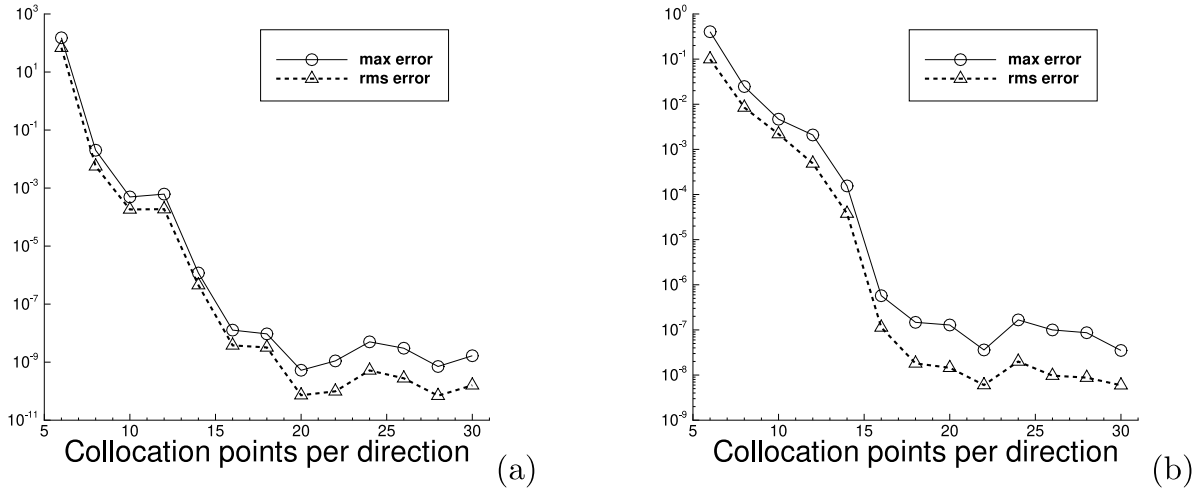

Fig. 3. Poisson equation: Maximum/rms errors of the VarPro solution versus the number of collocation points per direction (Q_1) obtained with (a) the cos activation function, and (b) the Gaussian activation function. Neural network [2, 200, 1] in (a,b); Q_1 is varied in (a,b); $\delta = 7.0$ in (a), and $\delta = 1.0$ in (b).

Let us first consider the VarPro results obtained with neural networks containing a single hidden layer. Fig. 2(b) shows the distribution of the absolute error of the VarPro solution in the xy plane. This result corresponds to the neural network architecture [2, 125, 1], with the cos activation function, a uniform set of $Q = 15 \times 15$ training collocation points, and $\delta = 7.0$ in Algorithm 3. The VarPro solution is highly accurate, with a maximum error around 10^{-8} in the domain.

Fig. 3 illustrates the convergence behavior of the VarPro solution as a function of the number of training collocation points in the domain. Here we employ a neural network [2, 200, 1], with the cos and the Gaussian activation functions. The number of collocation points in each direction (Q_1) is varied systematically. Fig. 3 shows the maximum and the rms errors of the VarPro solution in the domain as a function of Q_1, obtained using the cos activation function (plot (a)) and the Gaussian activation function (plot (b)). The VarPro errors decrease approximately exponentially when Q_1 is below around 20, and then appear to stagnate as Q_1 further increases. The VarPro errors reach a level around $10^{-11} \sim 10^{-9}$ with the cos activation function and a level around $10^{-9} \sim 10^{-7}$ with the Gaussian activation function.

Fig. 4 illustrates the convergence behavior of the VarPro accuracy with respect to the number of nodes in the hidden layer (M) of the network. Here we consider neural networks with the architecture [2, M, 1], where M is varied systematically, with the cos and Gaussian activation functions. A fixed uniform set of $Q = 21 \times 21$ training collocation points is used. Fig. 4 shows the maximum/rms errors of the VarPro solution in the domain as a function of M, obtained with the cos (plot(a)) and the Gaussian (plot (b)) activation functions. One can observe an approximately exponential decrease in the VarPro errors with increasing M (when M is below a certain value), and then the errors appear to stagnate (or increase slightly) as M further increases.

Fig. 5 illustrates the VarPro solution using a neural network containing two hidden layers. Here we have employed a neural network with the architecture [2, 20, 100, 1] and the cos activation function for all the hidden nodes. Fig. 5(a) shows the distribution of the absolute error of the VarPro solution obtained with a set of $Q = 18 \times 18$ uniform collocation points in the domain. The maximum error is on the level 10^{-9}, indicating a high accuracy. Fig. 5(b) depicts the maximum/rms errors of the VarPro solution as a function of the number of collocation points in each direction (Q_1). One can again observe an exponential decrease in the errors (before saturation) with increasing number of collocation points. All these results suggest that the VarPro method produces highly accurate results for solving the Poisson equation.

Table 2 compares the errors of the current VarPro method and the ELM method [21,25] for solving the Poisson equation. We have considered two neural networks having the architecture [2, M, 1], with $M = 100$ and $M = 200$. A uniform set of $Q = Q_1 \times Q_1$ training collocation points are employed on the domain, where Q_1 is varied between $Q_1 = 5$ and $Q_1 = 30$. In ELM the hidden-layer coefficients are set (and fixed) to uniform random values generated

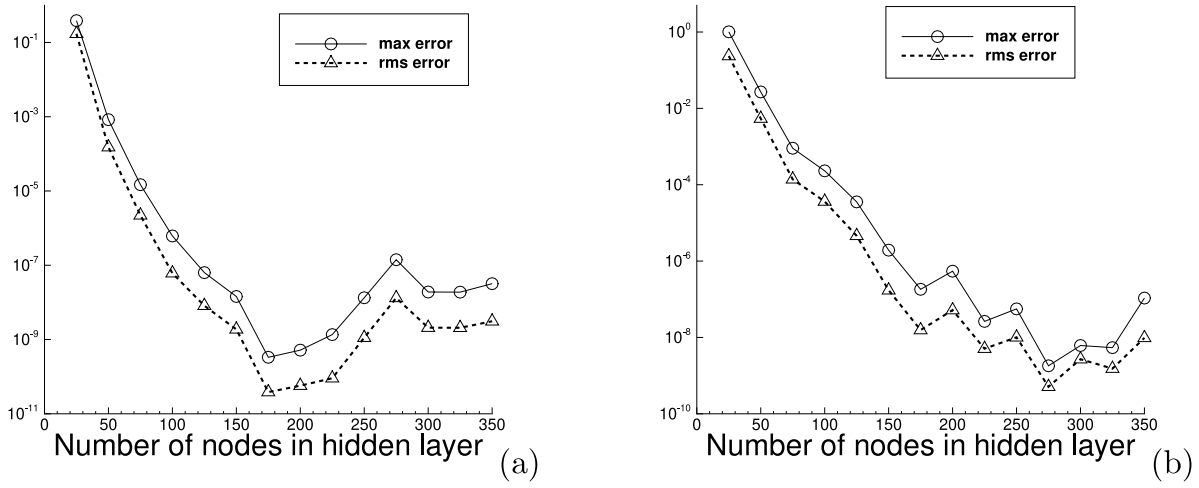

Fig. 4. Poisson equation: The maximum/rms errors of the VarPro solution versus the number of nodes in the hidden layer (M), computed using (a) the cos and (b) the Gaussian activation functions. Neural network [2, M, 1], where M is varied in (a,b); $Q = 21 \times 21$ in (a,b); $\delta = 7.0$ in (a) and $\delta = 2.0$ in (b).

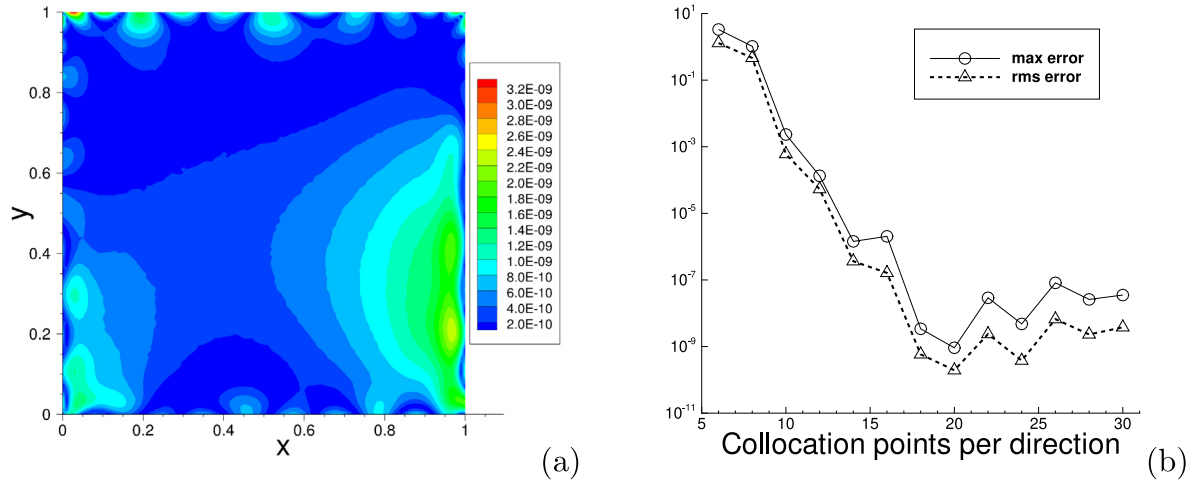

Fig. 5. Poisson equation (2 hidden layers in neural network): (a) Error distribution of the VarPro solution. (b) The maximum/rms errors of the VarPro solution versus the number of collocation points per direction (Q_1). Neural network [2, 20, 100, 1], with the cos activation function. $Q = 18 \times 18$ in (a), and is varied in (b). $\delta = 0.5$ in (a,b).

on $[-R_m, R_m]$, and in VarPro the hidden-layer coefficients are initialized (i.e. the initial guess $\boldsymbol{\theta}_0$ in Algorithm 3) to the same random values from $[-R_m, R_m]$. So the random hidden-layer coefficients in ELM and the initial hidden-layer coefficients in VarPro are identical. We have considered two R_m values, $R_m = 1.0$ and $R_m = R_{m0}$, where R_{m0} is the optimal R_m for ELM computed using the method from [25] and in this case $R_{m0} = 6.0$. We can make the following observations:

- The VarPro method in general produces considerably more accurate results than ELM, under the same settings and conditions, especially when the size of the neural network is still not quite large.
- The ELM accuracy has a fairly strong dependence on the R_m value. On the other hand, the VarPro accuracy is less sensitive or insensitive to the R_m value.

In these tests the VarPro method has produced errors on the order $10^{-8} \sim 10^{-10}$ with the given neural networks. We should point out that the ELM method can also achieve numerical errors on such levels, but it requires neural networks with a larger number of nodes in the hidden layer.

Table 2
Poisson equation: comparison of the maximum/rms errors obtained using the VarPro and ELM methods. cos activation function. In both VarPro and ELM, the hidden-layer coefficients are initialized/set to uniform random values generated on $[-R_m, R_m]$, with $R_m = 1.0$ or with $R_m = R_{m0}$. R_{m0} is the optimal R_m for ELM computed using the method from [25], and in this case $R_{m0} = 6.0$. $\delta = 5.0$ in VarPro.

Neural network	$[-R_m, R_m]$	Collocation points	VarPro		ELM	
			max-error	rms-error	max-error	rms-error
[2, 100, 1]	$R_m = 1$	5 × 5	6.470E+1	2.422E+1	1.247E+2	2.745E+1
		10 × 10	8.388E−3	3.941E−3	1.402E+1	2.575E+0
		15 × 15	6.018E−7	8.241E−8	1.475E+1	1.938E+0
		20 × 20	3.693E−7	4.216E−8	1.690E+1	2.527E+0
		25 × 25	5.845E−7	8.054E−8	1.777E+1	2.752E+0
		30 × 30	2.688E−7	2.867E−8	1.864E+1	2.916E+0
	$R_m = R_{m0} = 6$	5 × 5	2.156E+0	6.114E−1	2.156E+0	6.114E−1
		10 × 10	7.735E−4	1.497E−4	1.353E−1	2.497E−2
		15 × 15	4.175E−7	4.614E−8	3.019E−1	6.004E−2
		20 × 20	1.753E−7	1.974E−8	3.859E−1	7.575E−2
		25 × 25	8.443E−7	1.227E−7	4.336E−1	8.489E−2
		30 × 30	8.709E−8	1.088E−8	4.673E−1	9.157E−2
[2, 200, 1]	$R_m = 1$	5 × 5	5.948E+0	2.102E+0	9.325E+1	2.024E+1
		10 × 10	2.127E−2	4.398E−3	4.417E+0	7.083E−1
		15 × 15	6.082E−8	1.983E−8	5.615E+0	8.019E−1
		20 × 20	1.459E−9	1.203E−10	4.979E+0	7.610E−1
		25 × 25	1.782E−7	7.978E−8	5.077E+0	8.565E−1
		30 × 30	3.000E−9	3.420E−10	5.633E+0	8.804E−1
	$R_m = R_{m0} = 6$	5 × 5	7.292E−1	2.733E−1	7.292E−1	2.732E−1
		10 × 10	1.283E−4	3.705E−5	1.283E−4	3.705E−5
		15 × 15	2.315E−9	4.746E−10	9.822E−6	7.481E−7
		20 × 20	3.449E−10	3.722E−11	1.245E−5	1.518E−6
		25 × 25	6.379E−9	4.980E−10	1.174E−5	1.677E−6
		30 × 30	4.221E−10	4.086E−11	1.242E−5	1.800E−6

3.1.2. Advection equation

As another linear example we consider the spatial–temporal domain $\Omega = \{(x, t) \mid x \in [0, 3],\ t \in [0, t_f]\}$ in this test, where the temporal dimension t_f is to be specified below. We consider the initial/boundary value problem with the advection equation on Ω,

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0, \tag{26a}$$

$$u(0, t) = u(3, t), \tag{26b}$$

$$u(x, 0) = \sin\frac{2\pi}{3}(x - 2), \tag{26c}$$

where $u(x, t)$ is the field function to be solved for, and $c = -2.0$ is the wave speed. This problem has the following exact solution,

$$u(x, t) = \sin\frac{2\pi}{3}(x - 2t - 2). \tag{27}$$

Fig. 6(a) shows the distribution of this exact solution in the spatial–temporal domain with $t_f = 10$.

To solve this problem with the VarPro method, we employ a feed-forward neural network with one or two hidden layers, with the architecture given by $[2, M, 1]$ or $[2, 10, M, 1]$, where M is varied systematically in the tests. The two input nodes represent the spatial/temporal coordinates (x, t), and the output node represents the solution field $u(x, t)$. We employ the Gaussian function, $\sigma(x) = e^{-x^2}$, or the Gaussian error linear unit (GELU) [74], $\sigma(x) = \frac{1}{2}x\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$, as the activation function for all the hidden nodes. The output layer is linear and with zero bias.

We primarily consider a temporal dimension $t_f = 10$ for the domain Ω. We employ the block time marching (BTM) scheme from [21] together with the VarPro method for this problem; see Remark 2.5. We employ 10 uniform

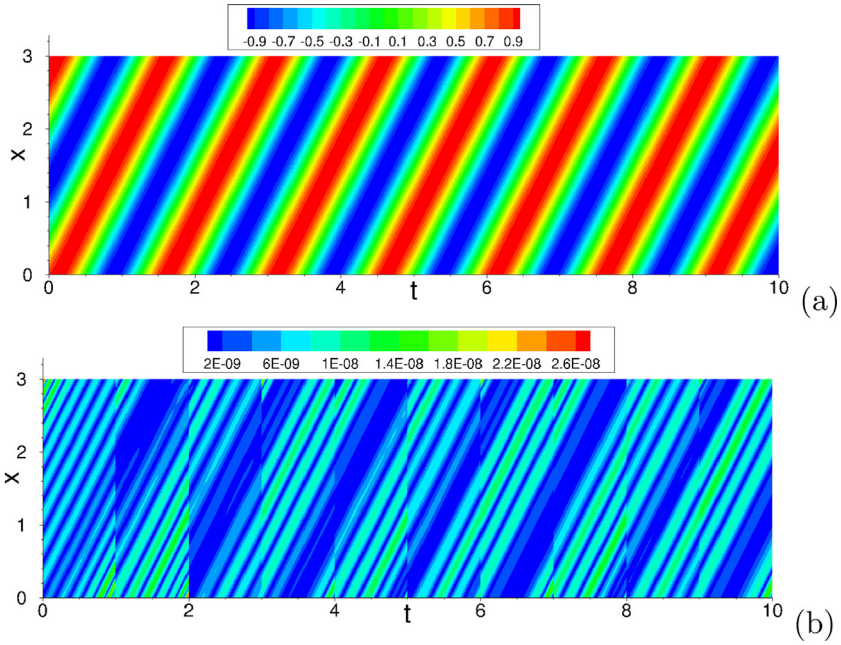

Fig. 6. Advection equation: Distributions of (a) the exact solution, and (b) the absolute error of the VarPro solution, in the spatial–temporal plane. $t_f = 10$. In (b), 10 time blocks, $Q = 21 \times 21$ uniform collocation points per time block, neural network [2, 100, 1], Gaussian activation function, the max sub-iterations is 2 and $\delta = 3.0$ in VarPro.

Table 3
Advection equation: main simulation parameters of the VarPro method.

Parameter	Value	Parameter	Value
t_f	10, or 100	Number of time blocks	10, or 100
Neural network	$[2, M, 1]$, or $[2, 10, M, 1]$	Training points Q	$Q_1 \times Q_1$
M	Varied	Q_1	Varied
Activation function	Gaussian, GELU	Testing points	$Q_2 \times Q_2$
Random seed	10	Q_2	101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	R_m	1.0
δ (Algorithm 3)	0.0, 0.05, 1.0, or 3.0	p (Algorithm 3)	0.5
max-subiterations	0, or 2	Threshold (Algorithm 3)	$1E{-}12$

time blocks in time and in each time block employ a uniform set of $Q = Q_1 \times Q_1$ training collocation points with the VarPro method, where Q_1 is varied systematically. Following Section 3.1.1, we employ a much larger uniform set of $Q_2 \times Q_2$ grid points within each time block to evaluate the trained neural network for the solution field and compute its errors by comparing with the exact solution (27). We have also considered another spatial–temporal domain with a much larger temporal dimension $t_f = 100$. Correspondingly, 100 uniform time blocks are employed in simulations of this case. The main simulation parameters for this problem are summarized in Table 3.

Let us first look into the VarPro errors obtained using neural networks with one hidden layer. Fig. 6(b) shows the distribution of the absolute error of the VarPro result in the spatial–temporal domain. This result is for the temporal dimension $t_f = 10$, and is obtained using a neural network [2, 100, 1] with the Gaussian activation function and a uniform set of $Q = 21 \times 21$ training collocation points in the domain. The VarPro result is highly accurate, with a maximum error on the order 10^{-8} in the overall domain.

Fig. 7 illustrates the convergence behavior of the VarPro solution with respect to the number of collocation points per direction (Q_1) in each time block. This is for the temporal dimension $t_f = 10$ computed with a neural network [2, 100, 1] and 10 time blocks. The plot (a) shows the maximum and rms errors in the overall domain of the VarPro solution as a function of Q_1 obtained with the Gaussian activation function. The plot (b) shows the corresponding

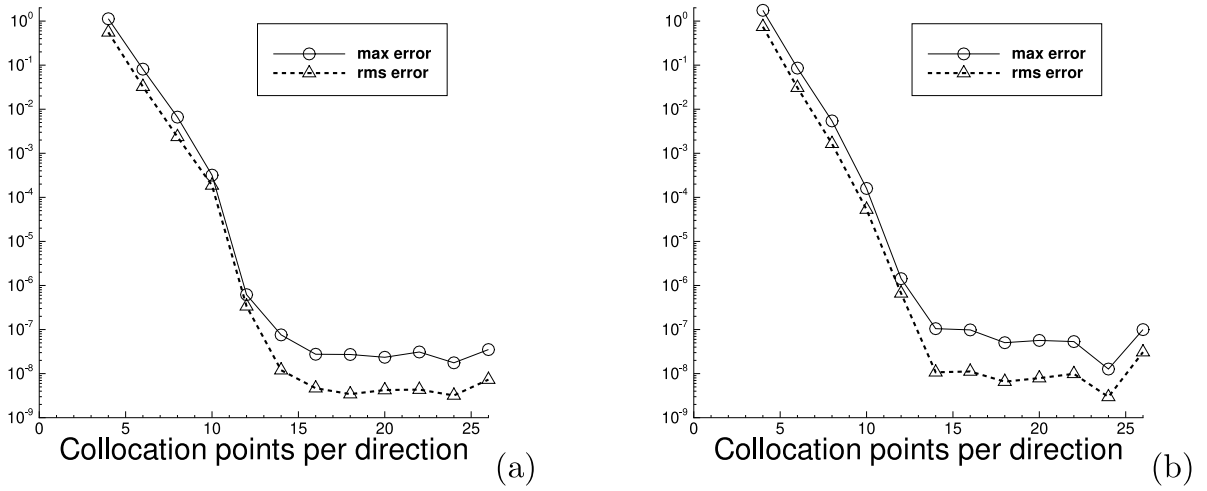

Fig. 7. Advection equation: the maximum/rms errors of the VarPro solution versus the number of collocation points per direction in each time block, obtained with (a) the Gaussian and (b) the GELU activation functions. In (a,b), $t_f = 10$, 10 time blocks, neural network [2, 100, 1], max-subiterations $= 2$ and $\delta = 1.0$ in VarPro.

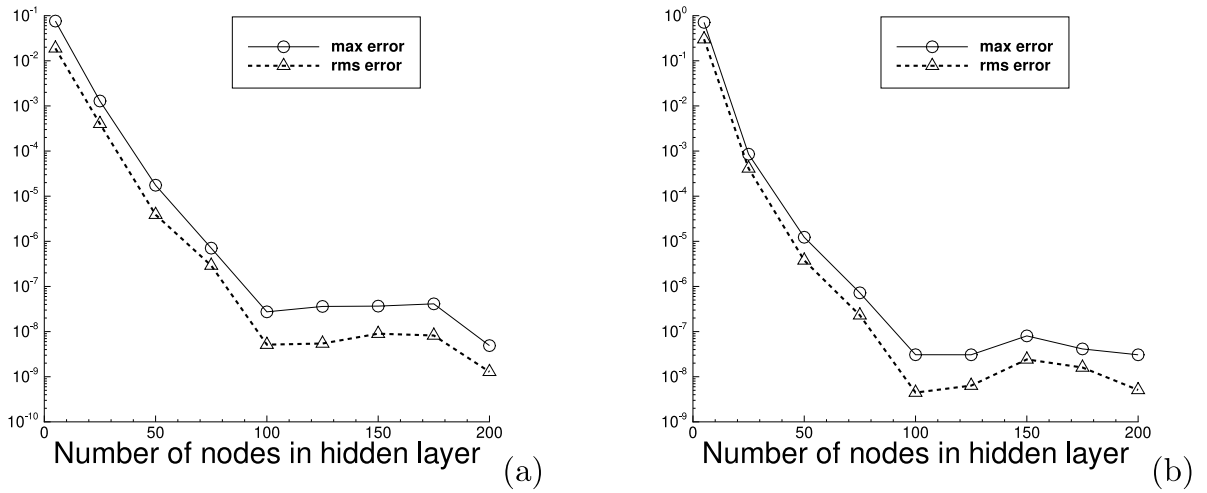

Fig. 8. Advection equation: the maximum/rms errors of the VarPro solution versus the number of nodes in the hidden layer (M) obtained with (a) the Gaussian and (b) the GELU activation functions. In (a,b), $t_f = 10$, 10 time blocks, $Q = 21 \times 21$ uniform collocation points, neural network $[2, M, 1]$ with M varied, max-subiterations $= 2$ and $\delta = 1.0$ in VarPro.

result obtained with the GELU activation function. The exponential decrease in the errors with increasing number of collocation points (before saturation) is unmistakable.

Fig. 8 illustrates the convergence behavior of the VarPro solution with respect to the number of nodes in the hidden layer (M). Here the domain corresponds to $t_f = 10$, with 10 time blocks and a uniform set of $Q = 21 \times 21$ training collocation points per time block in the VarPro simulation. The neural network is given by $[2, M, 1]$, where M is varied systematically. Fig. 8(a) and (b) shows the maximum/rms errors in the overall domain as a function of M obtained using the Gaussian and the GELU activation functions, respectively. The exponential decrease in the errors with increasing M (before saturation) is evident.

Fig. 9 illustrates the VarPro results computed using a neural network containing two hidden layers. The domain corresponds to $t_f = 10$, and the neural network has the architecture $[2, 10, 100, 1]$ with the Gaussian activation function. Fig. 9(a) shows the VarPro error distribution in the overall spatial–temporal plane, obtained with a uniform $Q = 21 \times 21$ training collocation points per time block. Fig. 9(b) depicts the maximum/rms VarPro errors in the

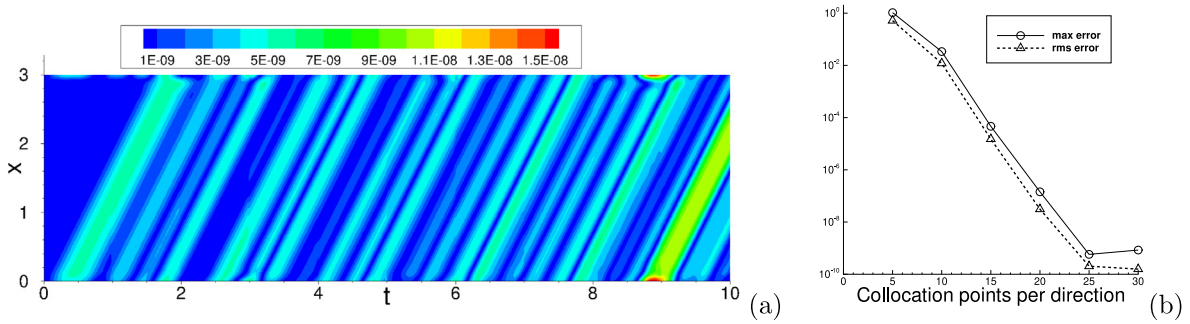

Fig. 9. Advection equation (two hidden layers in NN): (a) Error distribution of the VarPro solution in the spatial–temporal plane, and (b) the maximum/rms errors of the VarPro solution versus the number of collocation points per direction, obtained with 2 hidden layers in the neural network. In (a,b), $t_f = 10$, 10 uniform time blocks, neural network [2, 10, 100, 1], Gaussian activation function, max-subiterations = 2 and $\delta = 0.05$ in VarPro. $Q = 21 \times 21$ in (a) and is varied in (b).

Table 4
Advection equation: comparison of the maximum/rms errors of the solutions obtained using the VarPro and ELM methods. $t_f = 10$, 10 time blocks, Neural network [2, 100, 1], Gaussian activation function. In VarPro, the max-subiterations is 2 and $\delta = 1.0$. In ELM/VarPro, the hidden-layer coefficients are set/initialized to uniform random values from $[-R_m, R_m]$, with $R_m = 1.0$ or with $R_m = R_{m0} = 0.7$.

$[-R_m, R_m]$	Collocation	VarPro		ELM	
	points	max-error	rms-error	max-error	rms-error
$R_m = 1$	5×5	$2.505E{-}1$	$1.005E{-}1$	$2.505E{-}1$	$1.005E{-}1$
	10×10	$3.194E{-}4$	$1.864E{-}4$	$3.758E{-}4$	$1.245E{-}4$
	15×15	$4.254E{-}8$	$6.954E{-}9$	$4.232E{-}5$	$1.269E{-}5$
	20×20	$2.348E{-}8$	$4.240E{-}9$	$4.618E{-}5$	$1.402E{-}5$
	25×25	$1.493E{-}8$	$2.889E{-}9$	$5.471E{-}5$	$1.598E{-}5$
	30×30	$1.354E{-}8$	$2.528E{-}9$	$6.419E{-}5$	$1.773E{-}5$
$R_m = R_{m0} = 0.7$	5×5	$1.038E{-}1$	$3.252E{-}2$	$1.038E{-}1$	$3.252E{-}2$
	10×10	$1.996E{-}4$	$9.437E{-}5$	$2.240E{-}4$	$6.519E{-}5$
	15×15	$7.934E{-}8$	$9.495E{-}9$	$3.881E{-}5$	$9.642E{-}6$
	20×20	$9.261E{-}8$	$3.253E{-}8$	$3.471E{-}5$	$1.060E{-}5$
	25×25	$3.444E{-}8$	$4.931E{-}9$	$3.314E{-}5$	$1.117E{-}5$
	30×30	$4.670E{-}8$	$7.428E{-}9$	$3.283E{-}5$	$1.164E{-}5$

overall domain as a function of the collocation points per direction in each time block, demonstrating the exponential convergence behavior.

A comparison between the current VarPro method and the ELM method [21,25] for solving the advection equation is provided in Table 4. The maximum and rms errors of the VarPro and the ELM methods obtained on a series of training collocation points are listed. These results are for the domain $t_f = 10$, with 10 time blocks in block time marching. We have employed a neural network [2, 100, 1] with the Gaussian activation function. The random hidden-layer coefficients in ELM and the initial hidden-layer coefficients in VarPro are generated by $R_m = 1$ and $R_m = R_{m0} = 0.7$. It is evident that VarPro generally leads to significantly more accurate results than ELM.

Figs. 10 and 11 illustrate a longer-time simulation of the advection equation using the VarPro method and the block time marching scheme. Here the domain corresponds to $t_f = 100$. We have employed 100 uniform time blocks, a set of $Q = 25 \times 25$ uniform collocation points per time block, and a neural network [2, 150, 1] with the Gaussian activation function. Fig. 10(a) and (b) show the distributions of the VarPro solution and its absolute errors in the overall spatial–temporal plane. It can be observed that the VarPro method has produced highly accurate results, with the maximum error on the order 10^{-8} in this long-time simulation. Fig. 11(a) compares the time histories of the VarPro solution and the exact solution (27) at the mid-point of the domain ($x = 1.5$), and Fig. 11(b) shows the corresponding VarPro error history at this point. These results indicate that the VarPro method together with the block time marching scheme can produce highly accurate results in long-time simulations.

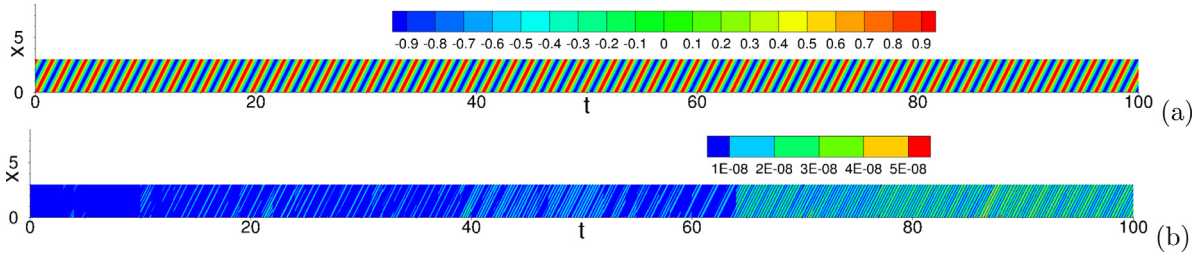

Fig. 10. Advection equation (long-time simulation): Distributions of (a) the VarPro solution and (b) its absolute error in the spatial–temporal domain. Domain: $(x, t) \in [0, 3] \times [0, 100]$, 100 uniform time blocks, $Q = 25 \times 25$ uniform collocation points per time block, neural network [2, 150, 1], Gaussian activation function, no subiteration (max-subiterations = 0) in VarPro.

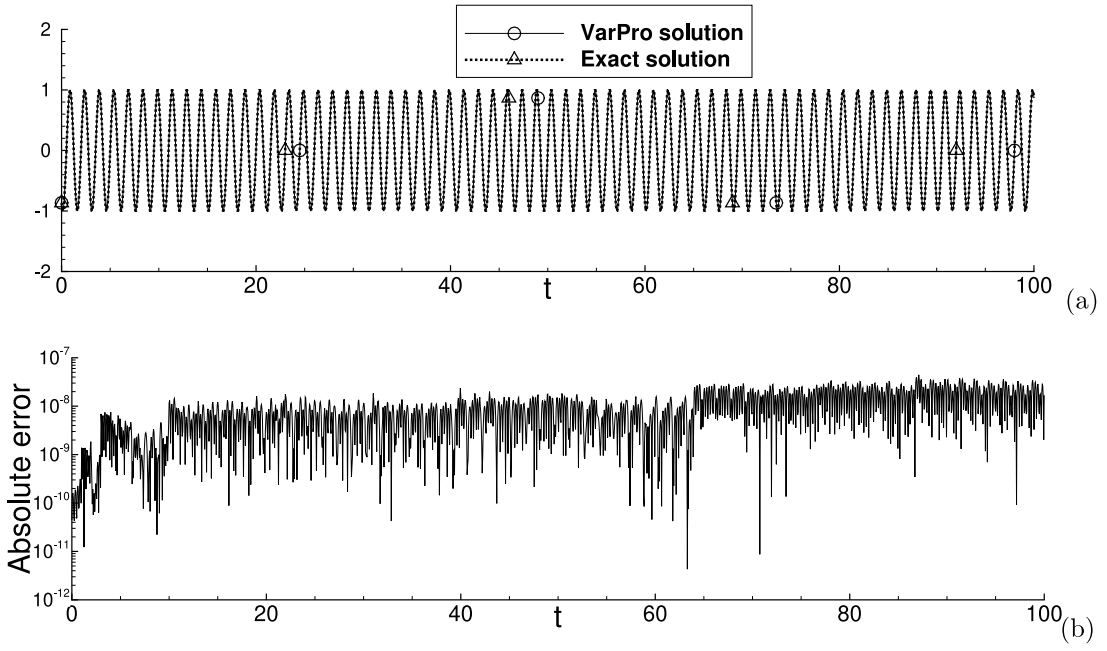

Fig. 11. Advection equation (long-time simulation): (a) Comparison of the time histories between the VarPro solution and the exact solution at the mid-point of the domain ($x = 1.5$). (b) Time history of the absolute error of the VarPro solution at the mid-point of the domain ($x = 1.5$). Simulation parameters and configurations here follow those of Fig. 10.

3.2. Nonlinear examples

3.2.1. Nonlinear Helmholtz equation

In the first nonlinear example, we consider the boundary value problem with a nonlinear Helmholtz equation on the unit square domain $[0, 1] \times [0, 1]$,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - 100u + 5\cos(2u) = f(x, y), \tag{28a}$$

$$u(x, 0) = g_1(x), \quad u(x, 1) = g_2(x), \quad u(0, y) = g_3(y), \quad u(1, y) = g_4(y), \tag{28b}$$

where $u(x, y)$ is the field function to be solved for, $f(x, y)$ is a prescribed source term, and g_i $(1 \leqslant i \leqslant 4)$ denote the boundary data. With f and g_i $(1 \leqslant i \leqslant 4)$ chosen appropriately, this problem admits the following analytic solution,

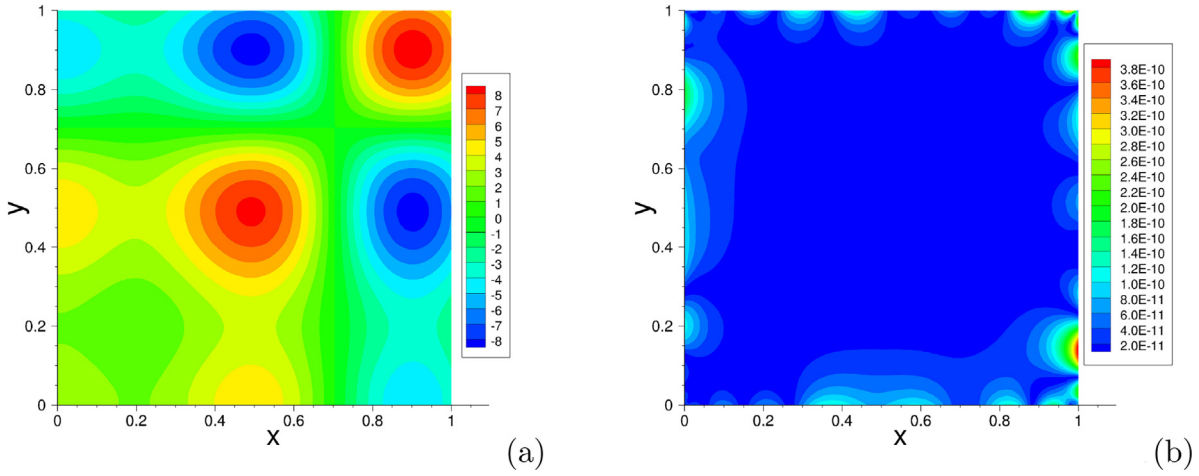

Fig. 12. Nonlinear Helmholtz equation: Distributions of (a) the exact solution and (b) the absolute error of the VarPro solution. In (b), neural network [2, 200, 1], "sin" activation function, $Q = 21 \times 21$ uniform collocation points, $\delta = 0.05$ in VarPro.

Table 5
Nonlinear Helmholtz equation: main simulation parameters of the VarPro method.

Parameter	Value	Parameter	Value
Neural network	$[2, M, 1]$, or $[2, 5, M, 1]$	Training points Q	$Q_1 \times Q_1$
M	Varied	Q_1	Varied
Activation function	sin, Gaussian	Testing points	$Q_2 \times Q_2$
Random seed	1	Q_2	101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	R_m	1.0
δ (Algorithm 3)	0.02, 0.05, 0.1, or 0.2	p (Algorithm 3)	0.5
max-subiterations	2	Threshold (Algorithm 3)	$1E{-}12$
max-iterations-newton	20	Tolerance-newton	$1E{-}8$

$$u = \left[\frac{5}{2}\cos\left(\frac{3}{2}\pi x - \frac{2}{5}\pi\right) + \frac{3}{2}\cos\left(3\pi x + \frac{3\pi}{10}\right) + \frac{1}{2}(e^{x} - e^{-x})\right]\left[\frac{5}{2}\cos\left(\frac{3}{2}\pi y - \frac{2}{5}\pi\right) + \frac{3}{2}\cos\left(3\pi y + \frac{3\pi}{10}\right) + \frac{1}{2}(e^{y} - e^{-y})\right]. \tag{29}$$

We employ this analytic solution in the following tests. Fig. 12(a) shows the distribution of this analytic solution in the xy plane.

We employ neural networks with the architectures $[2, M, 1]$ and $[2, 5, M, 1]$ in the VarPro simulations, where M is varied in the tests. The sine function, $\sigma(x) = \sin(x)$, or the Gaussian function, $\sigma(x) = e^{-x^2}$, is employed as the activation functions for the hidden nodes. A uniform set of $Q = Q_1 \times Q_1$ training collocation points, where Q_1 is varied, is used to train the neural network. The VarPro solution is computed on a larger set of $Q_2 \times Q_2$ (with $Q_2 = 101$) uniform grid points by evaluating the trained neural network, and compared with the analytic solution to compute its errors. Table 5 provides the main simulation parameters for this problem and the VarPro method. In this table "max-iterations-newton" denotes the maximum number of Newton iterations, and "tolerance-newton" denotes the relative tolerance for the Newton iteration (see lines 3 and 9 of Algorithm 4).

Fig. 12(b) illustrates the error distribution of a VarPro solution in the xy plane, computed using a neural network $[2, 200, 1]$ with the sin activation function and a uniform set of $Q = 21 \times 21$ collocation points. The result is observed to be highly accurate, with a maximum error on the order 10^{-10} in the domain.

Fig. 13 illustrates the convergence behavior of the VarPro method with respect to the number of training collocation points in the domain. In these tests the number of collocation points per direction (Q_1) is varied systematically. The two plots show the maximum/rms errors in the domain of the VarPro solution as a function of Q_1, obtained using the sin (plot (a)) and the Gaussian (plot (b)) activation functions. These VarPro results are attained

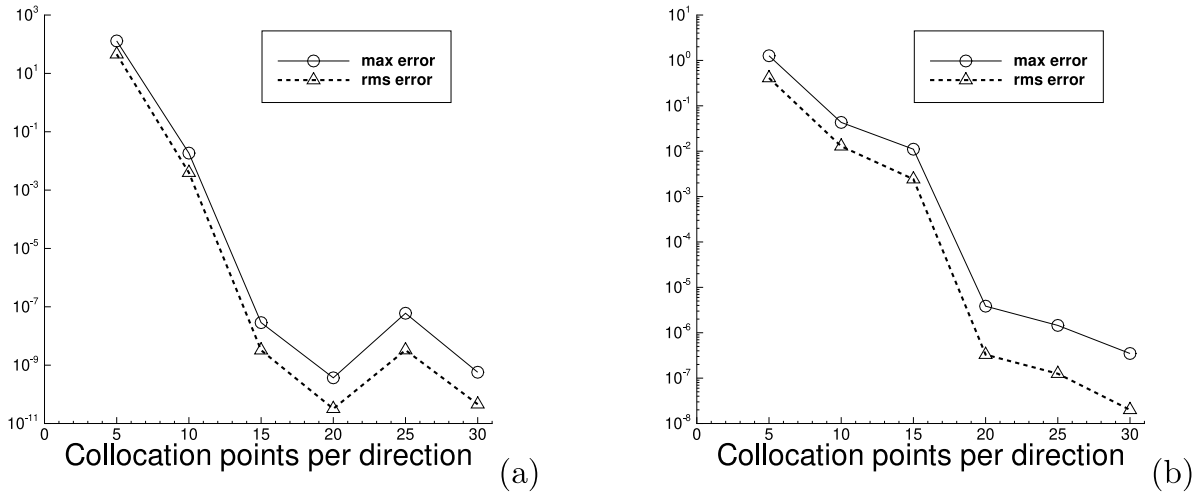

Fig. 13. Nonlinear Helmholtz equation: the maximum/rms errors of the VarPro solution versus the number of collocation points per direction, obtained using (a) the sine and (b) the Gaussian activation functions. Neural network [2, 200, 1], $\delta = 0.1$ in (a) and $\delta = 0.2$ in (b) with VarPro.

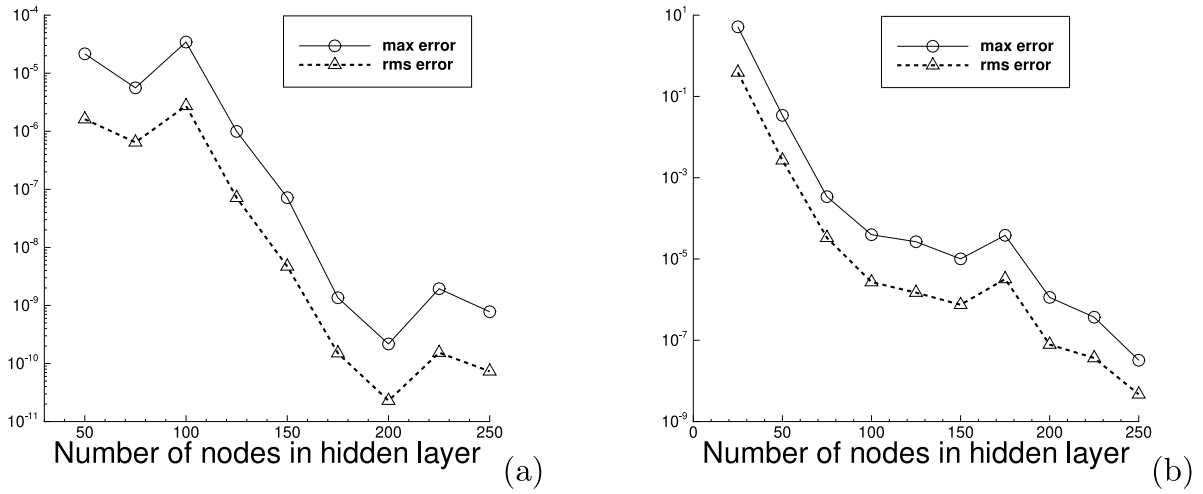

Fig. 14. Nonlinear Helmholtz equation: the maximum/rms errors of the VarPro solution versus the number of nodes in the hidden layer, obtained with (a) the sine and (b) the Gaussian activation functions. Neural network $[2, M, 1]$, where M is varied, $Q = 21 \times 21$, $\delta = 0.1$ in (a) and $\delta = 0.2$ in (b) with VarPro.

using a neural network [2, 200, 1]. We observe an exponential decrease in the VarPro errors (before saturation) with increasing number of collocation points.

Fig. 14 illustrates the convergence behavior of the VarPro solution with respect to the number of nodes in the hidden layer (M) for the nonlinear Helmholtz equation. Here the neural network has an architecture $[2, M, 1]$, where M is varied systematically, and a uniform set of $Q = 21 \times 21$ training collocation points is employed in the simulation. This figure shows the maximum/rms errors in the domain as a function of M, obtained using the sin (plot (a)) and the Gaussian (plot (b)) activation functions. The errors computed with the sin activation function appear not quite regular as M increases. But overall all these errors appear to decrease approximately exponentially with increasing M.

Fig. 15 illustrates the VarPro results obtained using two hidden layers in the neural network. Here we consider a neural network with the architecture $[2, 5, 200, 1]$, with the sin activation function. Fig. 15(a) shows the error distribution of the VarPro solution obtained using $Q = 20 \times 20$ training collocation points. In Fig. 15(b) the number

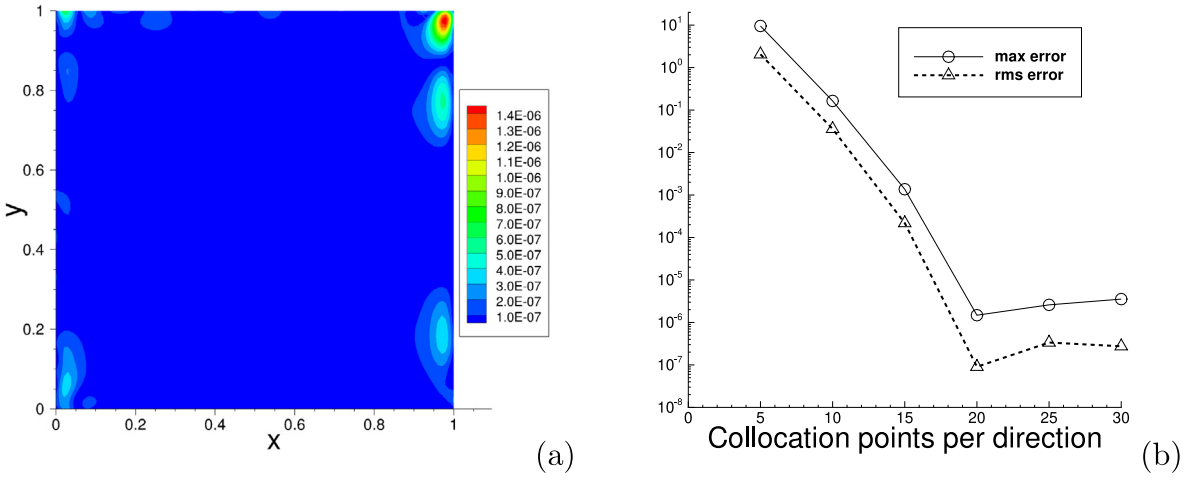

Fig. 15. Nonlinear Helmholtz equation (two hidden layers in NN): (a) Error distribution of the VarPro solution. (b) The VarPro maximum/rms errors versus the number of collocation points per direction (Q_1). Neural network [2, 5, 200, 1], sin activation function, $Q = 20 \times 20$ in (a) and is varied in (b), $\delta = 0.02$ in (a,b) with VarPro.

Table 6
Nonlinear Helmholtz equation: comparison of the maximum/rms errors of the VarPro and ELM solutions. Neural network [2, 200, 1], sin activation function. In VarPro, $\delta = 0.1$, and the tolerance-newton is set to $1E{-}8$ with $R_m = 1$ and to $1E{-}14$ with $R_m = 4.4$.

$[-R_m, R_m]$	Collocation points	VarPro		ELM	
		max-error	rms-error	max-error	rms-error
$R_m = 1$	5×5	$1.297E{+}2$	$4.536E{+}1$	$4.388E{+}0$	$8.195E{-}1$
	10×10	$1.855E{-}2$	$3.955E{-}3$	$7.701E{+}0$	$1.210E{+}0$
	15×15	$2.868E{-}8$	$3.252E{-}9$	$3.743E{-}1$	$4.767E{-}2$
	20×20	$3.679E{-}10$	$3.203E{-}11$	$1.280E{+}0$	$1.864E{-}1$
	25×25	$6.014E{-}8$	$3.281E{-}9$	$1.434E{+}0$	$2.198E{-}1$
	30×30	$5.709E{-}10$	$4.576E{-}11$	$8.365E{-}1$	$1.123E{-}1$
$R_m = R_{m0} = 4.4$	5×5	$6.742E{-}1$	$2.408E{-}1$	$1.182E{-}1$	$3.589E{-}2$
	10×10	$5.869E{-}3$	$1.169E{-}3$	$8.987E{-}7$	$1.861E{-}7$
	15×15	$7.130E{-}9$	$1.024E{-}9$	$6.690E{-}9$	$8.655E{-}10$
	20×20	$2.851E{-}9$	$2.364E{-}10$	$2.384E{-}8$	$2.835E{-}9$
	25×25	$4.193E{-}10$	$5.173E{-}11$	$3.133E{-}8$	$3.611E{-}9$
	30×30	$1.128E{-}9$	$1.029E{-}10$	$3.813E{-}8$	$4.194E{-}9$

of collocation points per direction (Q_1) is varied systematically, and the maximum/rms errors are plotted as a function of Q_1. An exponential decrease in the errors (before saturation) can be observed.

Table 6 is a comparison of the errors between the VarPro method and the ELM method for solving the nonlinear Helmholtz equation. These results are for a neural network [2, 200, 1] with the sin activation function. In ELM the random hidden-layer coefficients are set to, and in VarPro the hidden-layer coefficients are initialized to, uniform random values from $[-R_m, R_m]$, where $R_m = 1.0$ or $R_m = R_{m0} = 4.4$. Several sets of uniform training collocation points are tested, ranging from $Q = 5 \times 5$ to $Q = 30 \times 30$. The VarPro results are in general markedly more accurate than those of the ELM results. This is especially pronounced for those cases corresponding to $R_m = 1.0$.

3.2.2. Viscous Burgers' equation

In the second nonlinear example we use the viscous Burgers' equation to test the VarPro method. We will consider two solutions to the Burgers' equation: (i) a manufactured smooth solution, and (ii) an exact physical solution in which a sharp gradient develops in the domain over time.

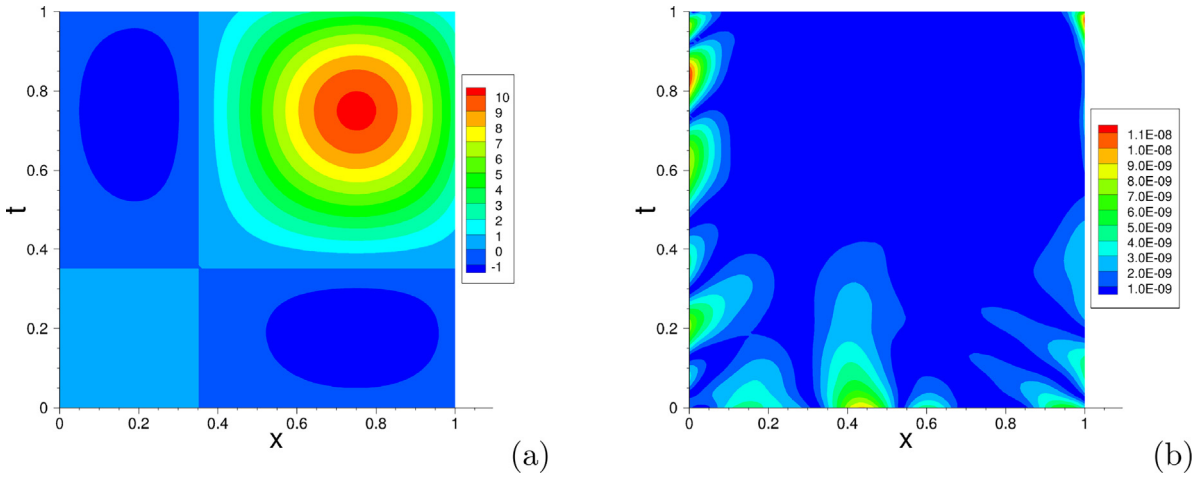

Fig. 16. Burgers' equation (with manufactured solution): Distributions of (a) the manufactured solution and (b) the absolute error of the VarPro solution in the spatial–temporal plane. In (b), neural network [2, 150, 1], and $Q = 31 \times 31$ training collocation points.

Let us first examine the convergence behavior of VarPro using a manufactured solution. Consider the spatial–temporal domain, $\Omega = \{(x,t) \mid x \in [0,1],\ t \in [0,1]\}$, and the following initial/boundary value problem on Ω,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = f(x,t), \tag{30a}$$

$$u(0,t) = g_1(t), \quad u(1,t) = g_2(t), \tag{30b}$$

$$u(x,0) = h(x). \tag{30c}$$

In the above equations, $u(x,t)$ is the field function to be solved for, $\nu = 0.05$, $f(x,t)$ is a prescribed source term, g_1 and g_2 are the boundary conditions, and h is the initial condition. We choose f, g_1, g_2 and h such that this problem has the following manufactured analytic solution,

$$u(x,t) = \left[2\cos\left(\pi x + \frac{2\pi}{5}\right) + \frac{3}{2}\cos\left(2\pi x - \frac{3\pi}{5}\right)\right]\left[2\cos\left(\pi t + \frac{2\pi}{5}\right) + \frac{3}{2}\cos\left(2\pi t - \frac{3\pi}{5}\right)\right]. \tag{31}$$

Fig. 16(a) shows the distribution of this manufactured solution in the spatial–temporal plane.

We employ neural networks with an architecture $[2, M, 1]$ in the VarPro simulations, where M is varied systematically in the tests. The two input nodes represent (x,t) and the linear output node represents the solution field $u(x,t)$. The Gaussian activation function, $\sigma(x) = e^{-x^2}$, is employed in all the hidden nodes. A uniform set of $Q = Q_1 \times Q_1$ collocation points in the spatial–temporal domain is used to train the neural network with the VarPro method, and Q_1 is varied systematically in the tests. The trained neural network is evaluated on a larger set of $Q_2 \times Q_2$ uniform grid points to attain the field solution, which is then compared with the analytic solution (31) to compute the errors. The main simulation parameters for this problem are summarized in Table 7.

Fig. 16(b) illustrates the distribution of the absolute error of a VarPro solution in the spatial–temporal plane. This is computed using a neural network [2, 150, 1] with a uniform set of $Q = 31 \times 31$ training collocation points. The VarPro solution can be observed to be quite accurate, with a maximum error on the order 10^{-8} in the domain.

Fig. 17 illustrates the convergence behavior of the VarPro method for solving the Burgers' equation. In these tests the neural network is given by $[2, M, 1]$, where M is either fixed at $M = 100$ or varied between $M = 25$ and $M = 250$. A set of $Q = Q_1 \times Q_1$ uniform training collocation points is used, where Q_1 is either fixed at $Q_1 = 31$ or varied between $Q_1 = 10$ and $Q_1 = 35$. Fig. 17(a) shows the maximum/rms errors of the VarPro solution as a function of Q_1, corresponding to a fixed $M = 100$ for the neural network. The error behavior is not quite regular. With a smaller Q_1 (e.g. 10 or 15) the errors are at a level $1 \sim 10$, while with a larger Q_1 (20 and beyond) the errors abruptly drop to a level around $10^{-8} \sim 10^{-6}$. We observe that with the smaller $Q_1 = 10$ and 15 the Newton iteration fails to converge to the prescribed tolerance within the prescribed maximum number of

Table 7
Burgers' equation: main simulation parameters of VarPro with the manufactured solution (31).

Parameter	Value	Parameter	Value
Domain	$(x, t) \in [0, 1] \times [0, 1]$	Block time marching	None
Neural network	$[2, M, 1]$	Training points Q	$Q_1 \times Q_1$
M	Varied	Q_1	Varied
Activation function	Gaussian	Testing points	$Q_2 \times Q_2$
Random seed	10	Q_2	101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	R_m	1.0
δ (Algorithm 3)	Un-used	p (Algorithm 3)	Un-used
max-subiterations	0 (no subiteration)	Threshold (Algorithm 3)	$1E{-}12$
max-iterations-newton	50	Tolerance-newton	$1E{-}8$

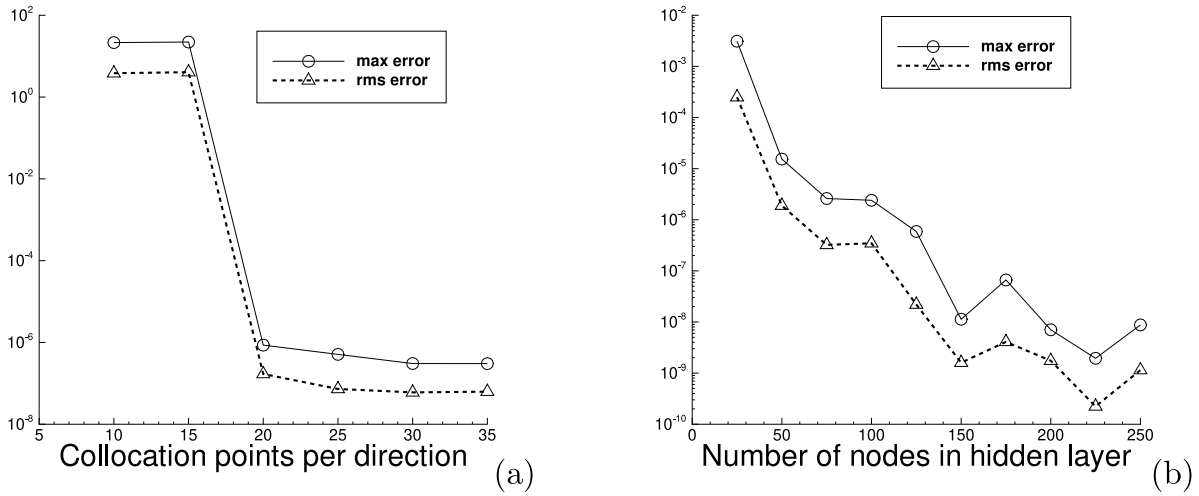

Fig. 17. Burgers' equation (with manufactured solution): The maximum/rms errors of the VarPro solution versus (a) the number of collocation points in each direction (Q_1), and (b) the number of nodes in the hidden layer (M) of the neural network. Neural network $[2, M, 1]$, $Q = Q_1 \times Q_1$ uniform collocation points. $M = 100$ in (a) and is varied in (b). $Q_1 = 31$ in (b) and is varied in (a).

iterations. Fig. 17(b) shows the maximum/rms errors as a function of M, corresponding to a fixed $Q = 31 \times 31$ for the collocation points. The errors can be observed to decrease approximately exponentially with increasing M.

Table 8 provides a comparison of the solution errors obtained using the VarPro method and the ELM method [21,25] for the Burgers' equation. These are computed using a fixed uniform set of $Q = 31 \times 31$ collocation points and a series of neural networks with the architecture $[2, M, 1]$, where M is varied between $M = 25$ and $M = 150$. The random hidden-layer coefficients in ELM are set, and the hidden-layer coefficients in VarPro are initialized, by using $R_m = 1$ and $R_m = R_{m0} = 0.9$ in the tests. It is evident that the VarPro method produces significantly more accurate results than the ELM method.

We next further test the VarPro method using an exact solution with a sharp gradient developed in the spatial–temporal domain. We consider the spatial–temporal domain, $\Omega_a = \{(x, t) \mid x \in [-1, 1],\ t \in [0, 1.05]\}$, and the following problem on Ω_a with the Burgers' equation,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0, \tag{32a}$$

$$u(-1, t) = 0, \quad u(1, t) = 0, \quad u(x, 0) = -\sin(\pi x), \tag{32b}$$

where $\nu = \frac{1}{100\pi}$. This initial boundary value problem has an exact solution given by [75],

$$u(x, t) = -\frac{\int_{-\infty}^{\infty} \sin\pi(x - \xi) F(x - \xi) e^{-\frac{\xi^2}{4\nu t}} d\xi}{\int_{-\infty}^{\infty} F(x - \xi) e^{-\frac{\xi^2}{4\nu t}} d\xi}, \quad \text{with } F(\xi) = e^{-\frac{\cos(\pi\xi)}{2\pi\nu}}. \tag{33}$$

Table 8
Burgers' equation (with manufactured solution): comparison of the maximum/rms errors of the VarPro and ELM solutions. Neural network $[2, M, 1]$ with M varied; fixed $Q = 31 \times 31$ training collocation points.

$[-R_m, R_m]$	M	VarPro		ELM	
		max-error	rms-error	max-error	rms-error
$R_m = 1$	25	$3.111E{-3}$	$2.499E{-4}$	$6.382E{+0}$	$8.382E{-1}$
	50	$1.538E{-5}$	$1.890E{-6}$	$6.669E{-2}$	$1.016E{-2}$
	75	$2.603E{-6}$	$3.222E{-7}$	$1.216E{-2}$	$1.406E{-3}$
	100	$2.406E{-6}$	$3.471E{-7}$	$4.189E{-4}$	$6.540E{-5}$
	125	$5.894E{-7}$	$2.193E{-8}$	$1.088E{-4}$	$1.099E{-5}$
	150	$1.131E{-8}$	$1.599E{-9}$	$4.387E{-6}$	$5.623E{-7}$
$R_m = R_{m0} = 0.9$	25	$1.592E{-3}$	$2.150E{-4}$	$3.501E{+0}$	$6.245E{-1}$
	50	$1.069E{-5}$	$1.794E{-6}$	$1.390E{-1}$	$7.607E{-3}$
	75	$3.049E{-6}$	$3.576E{-7}$	$1.948E{-2}$	$1.524E{-3}$
	100	$4.831E{-6}$	$3.311E{-7}$	$7.639E{-4}$	$4.780E{-5}$
	125	$4.163E{-7}$	$7.029E{-8}$	$5.864E{-5}$	$6.823E{-6}$
	150	$6.063E{-8}$	$7.242E{-9}$	$2.000E{-6}$	$2.877E{-7}$

Table 9
Burgers' equation (with exact solution): simulation parameters of the VarPro method.

Parameter	Value	Parameter	Value
Domain	$(x, t) \in [-1, 1] \times [0, 1.05]$	Block time marching	Yes
Time blocks	7 (uniform)	Sub-domains/time-block	6 (non-uniform)
Sub-domain boundaries	$x = 0, \pm 0.02, \pm 0.1, \pm 1$	Random seed	10
Local Neural network	$[2, 250, 1]$	Training points/sub-domain	21×21
Activation function	Gaussian	Testing points/sub-domain	101×101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	$R_m = R_{m0}$	$R_{m0} = 2.0$
δ (Algorithm 3)	Un-used	p (Algorithm 3)	Un-used
max-subiterations	0 (no subiteration)	Threshold (Algorithm 3)	$1E{-12}$
max-iterations-newton	10	Tolerance-newton	$1E{-8}$
C^k continuity	C^1 on sub-domain boundary		

Fig. 18(a) illustrates the distribution of this solution in the spatial–temporal plane, in which a sharp gradient can be observed to develop in the domain after around $t \approx 0.35$.

We solve this problem by the local VarPro method (see Remark 2.6) together with the block time marching scheme (see Remark 2.5). We partition Ω_a along the time t into 7 uniform time blocks, with a temporal dimension 0.15 for each time block, and compute the time blocks individually and successively. We decompose the spatial–temporal domain of each time block into 6 sub-domains (non-uniform) along the x direction, with the x coordinates of the sub-domain boundaries given by $x = 0, \pm 0.02, \pm 0.1$, and ± 1. C^1 continuity conditions for the local solution fields are imposed along the x direction across the interior sub-domain boundaries. On each sub-domain we employ a local neural network $[2, 250, 1]$ with the Gaussian activation function, where the two input nodes denote the (x, t) of the local sub-domain and the single output node denotes the solution $u(x, t)$ restricted to this sub-domain. We employ a uniform set of 21×21 collocation points on each sub-domain in the training of the overall neural network with the VarPro method. After the network is trained, we evaluate the neural network on a uniform set of 101×101 grid points on each sub-domain to obtain the VarPro solution data, which is then compared with the exact solution (33) evaluated on the same set of grid points to compute the point-wise errors of the VarPro solution on the overall spatial–temporal domain Ω_a. The main parameters for the configuration and the VarPro simulation of this problem is listed in Table 9. For the initial guess of the hidden-layer coefficients $\boldsymbol{\theta}_0$ in Algorithm 3, we have employed the uniform random values generated on the interval $[-R_m, R_m]$, with $R_m = R_{m0}$, where $R_{m0} = 2.0$ is the optimal R_m for the ELM method obtained using the method of [25] for the problem (32).

Fig. 18 summarizes the obtained VarPro solution to the problem (32). The plots (b) and (c) depict the distributions of the VarPro solution and its point-wise absolution error in the spatial–temporal domain, respectively. Visually one cannot discern any difference between the distributions of the VarPro solution (plot (b)) and the exact solution (plot (a)). The error distribution indicates that the solution obtained by the VarPro method is quite accurate. On

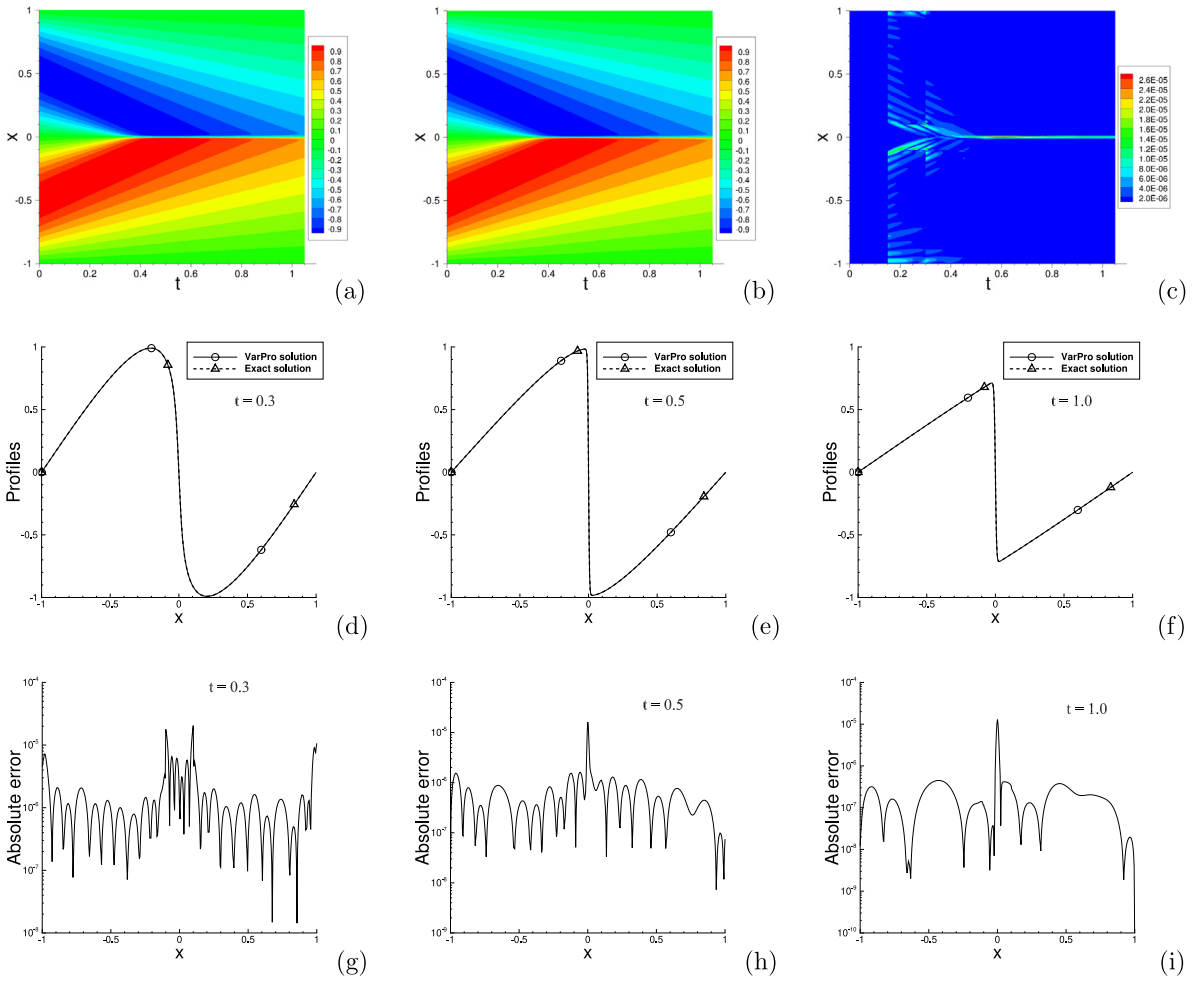

Fig. 18. Burgers' equation (with exact solution): Distributions of (a) the exact solution, (b) the VarPro solution, and (c) the absolute error of the VarPro solution in the spatial–temporal plane. Comparison of profiles of the VarPro and the exact solutions at (d) $t = 0.3$, (e) $t = 0.5$, and (f) $t = 1.0$. Error profiles of the VarPro solution at (g) $t = 0.3$, (h) $t = 0.5$, and (i) $t = 1.0$. See Table 9 for the simulation parameters.

the spatial–temporal domain Ω_a, the maximum error of the VarPro solution is 2.72×10^{-5} and the rms error is 4.47×10^{-6}. The plots (d,e,f) compare the profiles (along x) of the VarPro solution and the exact solution given by (33) at three time instants $t = 0.3$, 0.5, and 1.0, respectively. One can observe the sharp gradient of the solution in the middle of the spatial domain at the latter two instants, which is close to a jump discontinuity. It can also be observed that the VarPro profiles and the exact-solution profiles overlap with each other. The plots (g,h,i) show the corresponding error profiles of the VarPro solution at these three instants, illustrating the quite high accuracy of the VarPro result. The results of Fig. 18 suggest that, even if the solution field is not quite smooth, the VarPro method is still capable of producing simulation results with a fairly high accuracy.

3.2.3. Nonlinear Klein–Gordon equation

In the last example we use the nonlinear Klein–Gordon equation [76] to test the VarPro method. Consider the spatial–temporal domain, $\Omega = \{(x,t) \mid x \in [0,1],\ t \in [0,2]\}$, and the initial/boundary value problem with the nonlinear Klein–Gordon equation on Ω,

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} + u + \sin(u) = f(x,t), \tag{34a}$$

$$u(0,t) = g_1(t), \quad u(1,t) = g_2(t), \tag{34b}$$

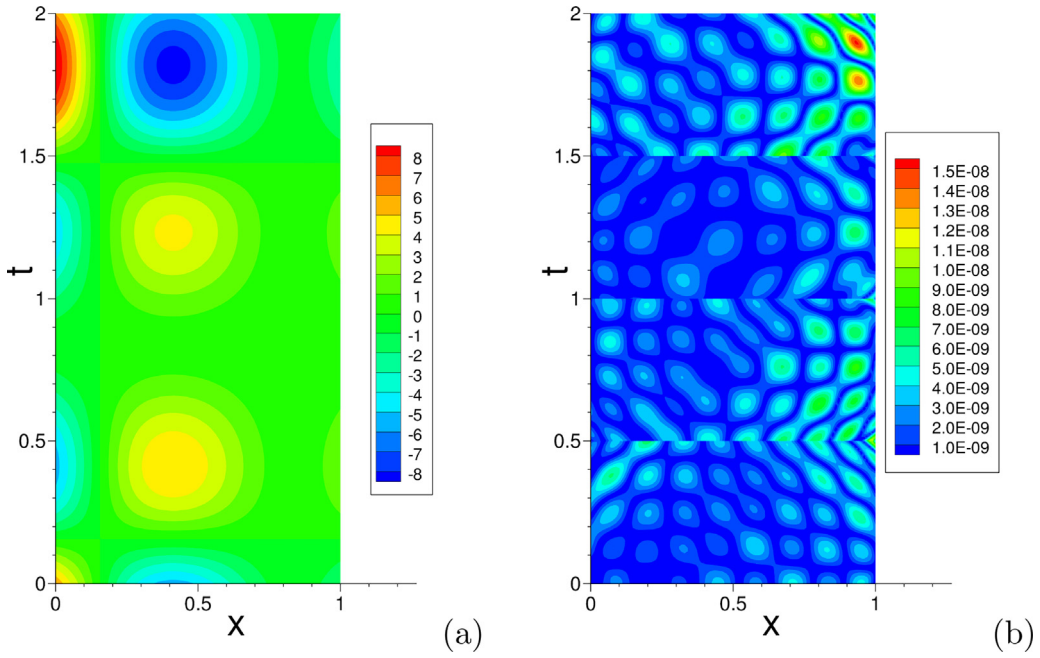

Fig. 19. Nonlinear Klein–Gordon equation: Distributions of (a) the exact solution and (b) the absolute error of the VarPro solution in the spatial–temporal plane. In (b), 4 uniform time blocks in domain, neural network $[2, 200, 1]$, $Q = 21 \times 21$ uniform collocation points per time block.

$$u(x,0) = h_1(x), \qquad \left.\frac{\partial u}{\partial t}\right|_{(x,0)} = h_2(x), \tag{34c}$$

where $u(x,t)$ is the field function to be solved for, $f(x,t)$ is a prescribed source term, g_1 and g_2 are the boundary conditions, and h_1 and h_2 are the initial conditions. We choose the source term, the boundary and initial conditions appropriately such that this problem has the following analytic solution,

$$u(x,t) = \left[2\cos\left(\pi x + \frac{\pi}{5}\right) + \frac{9}{5}\cos\left(2\pi x + \frac{7\pi}{20}\right)\right]\left[2\cos\left(\pi t + \frac{\pi}{5}\right) + \frac{9}{5}\cos\left(2\pi t + \frac{7\pi}{20}\right)\right]. \tag{35}$$

We employ this analytic solution to test the accuracy of the VarPro method. Fig. 19(a) shows the distribution of this analytic solution in the spatial–temporal plane.

We employ the block time marching scheme [21] together with the VarPro method to solve this problem. We use 4 uniform time blocks in the domain, and on each time block employ a neural network with the architecture $[2, M, 1]$, where M is varied in the tests. The two input nodes represent (x,t), and the linear output node represents the field solution $u(x,t)$. The Gaussian activation function $\sigma(x) = e^{-x^2}$ is employed for all the hidden nodes. On each time block a uniform set of $Q = Q_1 \times Q_1$ collocation points, where Q_1 is varied, is used to train the neural network. The trained neural network is evaluated on a larger set of $Q_2 \times Q_2$ uniform grid points to obtain the field solution $u(x,t)$, which is compared with the exact solution (35) to compute the errors of the VarPro simulation. Table 10 summarizes the main simulation parameters for this problem.

Fig. 19(b) shows the distribution of the absolute error of a VarPro simulation in the spatial–temporal plane. In this simulation the neural network architecture is given by $[2, 200, 1]$, and a uniform set of $Q = 21 \times 21$ training collocation points is used on each time block. The maximum error level is around 10^{-8} on the overall domain, indicating that the VarPro result is quite accurate.

Fig. 20 illustrates the convergence behavior of the VarPro method for solving the nonlinear Klein–Gordon equation. Here we employ the neural network $[2, M, 1]$, and $Q = Q_1 \times Q_1$ uniform collocation points in each time block. In the first group of tests we fix $M = 200$ and vary Q_1 systematically. In the second group of tests we fix $Q_1 = 31$ and vary M systematically. The maximum and rms errors of the VarPro solution in the overall domain are computed for each case. Fig. 20(a) shows these errors as a function of Q_1 for the first group of tests,

Table 10
Nonlinear Klein–Gordon equation: main simulation parameters of the VarPro method.

Parameter	Value	Parameter	Value
Domain	$(x, t) \in [0, 1] \times [0, 2]$	Time blocks	4
Neural network	$[2, M, 1]$	Training points Q	$Q_1 \times Q_1$
M	Varied	Q_1	Varied
Activation function	Gaussian	Testing points	$Q_2 \times Q_2$
Random seed	22	Q_2	101
Initial guess $\boldsymbol{\theta}_0$	Random values on $[-R_m, R_m]$	R_m	1.0
δ (Algorithm 3)	Un-used	p (Algorithm 3)	Un-used
max-subiterations	0 (no subiteration)	Threshold (Algorithm 3)	$1E{-}12$
max-iterations-newton	20	Tolerance-newton	$1E{-}8$

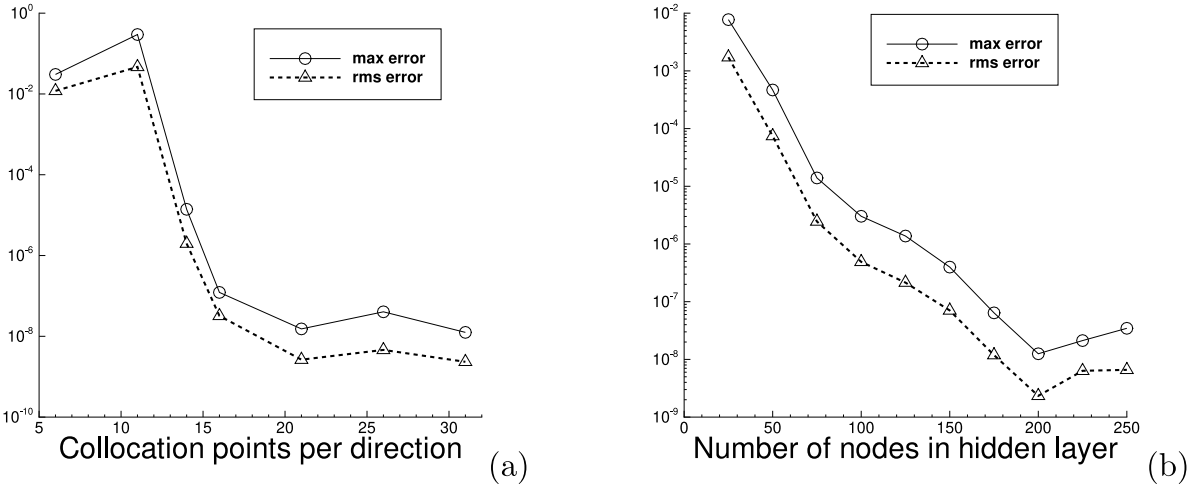

Fig. 20. Nonlinear Klein–Gordon equation: the maximum/rms errors of the VarPro solution in the overall domain versus (a) the number of collocation points per direction (Q_1) in each time block, and (b) the number of nodes in the hidden layer (M) of the neural network. In (a,b), neural network $[2, M, 1]$, with $Q = Q_1 \times Q_1$ training collocation points. $M = 200$ in (a) and is varied in (b). $Q_1 = 31$ in (b) and is varied in (a).

and Fig. 20(b) shows these errors as a function of M for the second group of tests. These results indicate that the VarPro errors decrease approximately exponentially with increasing number of collocation points or with increasing number of nodes in the hidden layer. We also notice some irregularity in the errors of Fig. 20(a) when the number of collocation points is small.

4. Concluding remarks

In this paper we have presented a variable projection-based method together with artificial neural networks for solving linear and nonlinear partial differential equations. The basic idea of variable projection (VarPro) is to distinguish the linear parameters from the nonlinear parameters, and then eliminate the linear parameters to attain a reduced formulation of the problem. One can then solve the reduced problem for the nonlinear parameters first, and then compute the linear parameters by the linear least squares method afterwards.

Approximating linear PDEs (with linear boundary/initial conditions) by variable projection and artificial neural networks is conceptually straightforward. In this case, in the resultant nonlinear least squares problem the output-layer coefficients are the linear parameters, and the hidden-layer coefficients are the nonlinear parameters. The output-layer coefficients are expressed in terms of the hidden-layer coefficients by solving a linear least squares problem, and they are eliminated from the problem. The reduced problem involves only the hidden-layer coefficients, and it is solved by a nonlinear least squares method. The main issues with the VarPro implementation lie in the computations of the residual function and the Jacobian matrix of the reduced problem. We have discussed in some

Table 11
Comparison of the computational cost (network training time) between VarPro and ELM for solving the advection equation (Section 3.1.2) and the nonlinear Helmholtz equation (Section 3.2.1). The hidden-layer coefficients in ELM are set/fixed to, and the hidden-layer coefficients in VarPro are initialized to, uniform random values from $[-R_m, R_m]$ with $R_m = 1.0$. The cases in this table are selected from and correspond to those cases in Tables 4 and 6. Please refer to the corresponding cases in Tables 4 and 6 for the VarPro/ELM errors.

Test problem	Neural network	Collocation Points	VarPro training time (s)	ELM training time (s)
Advection equation	[2, 100, 1]	10×10	7.4	0.29
		15×15	15.4	0.31
		20×20	78.2	0.32
Nonlinear Helmholtz equation	[2, 200, 1]	10×10	1.75	2.0
		15×15	36.5	2.8
		20×20	78.7	4.0

detail how to implement these computations with neural networks in Algorithms 1 and 2 and in Remarks 2.1 and 2.2.

For approximating nonlinear PDEs, or linear PDEs with nonlinear boundary/initial conditions, the variable projection approach cannot be directly used. This is because the resultant nonlinear least squares problem is not separable. In this case, all the weight/bias coefficients in the neural network become nonlinear parameters, even if the output layer contains no activation function.

To overcome this issue, we have presented a Newton/variable projection (Newton-VarPro) method for approximating nonlinear PDEs, or linear PDEs with nonlinear boundary/initial conditions. We first linearize the problem, with a particular linearized form for the Newton iteration. The linearization is formulated in terms of the updated approximation field, not the increment field. This is critical to the accuracy of the current Newton-VarPro method. The linearized system is then solved by the variable projection approach together with artificial neural networks. Therefore, for nonlinear PDEs our method involves an overall Newton iteration, and within each iteration the variable projection method is used to solve the linearized problem to attain the updated approximation field. Upon convergence of the Newton iteration, the solution to the nonlinear problem is represented by the weight/bias coefficients of the neural network.

We have presented ample numerical examples to test the accuracy of the variable projection method. It is observed that, for smooth field solutions, the errors of the VarPro method decrease exponentially or nearly exponentially with increasing number of collocation points or with increasing number of output-layer coefficients. The test results unequivocally show that the VarPro method is highly accurate. Even with a fairly small number of nodes in the neural network, or with a fairly small set of collocation points, the VarPro method can produce very accurate simulation results.

In particular, we have compared the current VarPro method with the extreme learning machine (ELM) method [25], which is arguably the most accurate neural network-based PDE solver so far [21,25]. Under the same simulation conditions and settings, VarPro generally leads to significantly more accurate results than ELM, especially in cases with a fairly small or a moderate number of nodes in the neural network.

While the VarPro method is significantly superior to ELM in terms of the accuracy, its computational cost (i.e. training time of the neural network) is generally much higher than that of the ELM method. This is because in VarPro one needs to solve the reduced problem for the hidden-layer coefficients by a nonlinear least squares computation, apart from the computation for the linear output-layer coefficients. In contrast, in ELM only the linear output-layer coefficients are computed, while the hidden-layer coefficients are randomly assigned and fixed. Table 11 illustrates this point with a list of the network training time for VarPro and for ELM with selected cases in solving the advection equation (Section 3.1.2) and the nonlinear Helmholtz equation (Section 3.2.1).

The variable projection method is a powerful technique for training artificial neural networks, providing a considerably superior accuracy for scientific machine learning, as demonstrated by the numerical examples in the current paper. The Newton-VarPro method developed herein provides an effective tool and enables the use of the variable projection strategy to tackle nonlinear problems in scientific machine learning. The application potential of this technique is enormous. This and related aspects, as well as further studies and improvements, of this technique will be pursued in a future endeavor.

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

This work was partially supported by NSF, United States (DMS-2012415).

Appendix A. Effect of seeds in random number generator on VarPro errors

As stated in the main text, we have employed fixed seed values in the random number generator (RNG) for those numerical experiments in Section 3, so that the reported results therein can be exactly reproducible. In this Appendix we set the seed of the RNG to random values, and study the effect of the random seeds on the VarPro errors for solving the Poisson equation (Section 3.1.1).

In Section 3.1.1 we have employed a fixed seed value 1 in the RNGs of the Tensorflow library and the numpy package. In the following tests, we employ the same seed value for the RNGs in Tensorflow and numpy, and this seed is set to be a random integer generated on the interval $[0, 10000)$ by a uniform distribution. The simulation parameters for VarPro and the problem configuration here follow those of Section 3.1.1.

Fig. 21 illustrates the l^∞ (i.e. maximum) errors and the l^2 (i.e. rms) errors of the VarPro solution from 10 random runs, and their statistics, with respect to increasing collocation points in the domain, for the cosine activation function. The numerical experiments, simulation parameters and the configuration here mirror those in Fig. 3(a). Fig. 21(a) shows the l^∞ errors as a function of the collocation points in each direction from the 10 random runs, with the corresponding random seeds given in the legend. Fig. 21(b) shows the corresponding l^2 errors from these random runs. Fig. 21(c) shows the statistics of the l^∞ error computed from the 10 random runs as shown in Fig. 21(a). For each given number of collocation points, the maximum, the minimum, the median, and the mean (average) of the l^∞ errors among the 10 random runs are computed and depicted in Fig. 21(c). Fig. 21(d) shows the statistical results for the l^2 error corresponding to those of Fig. 21(b). We can make the following observations from these results:

- The specific VarPro error values resulting from different random seeds are not the same. But their error levels are approximately comparable. Their values are distributed in a band around some mean or median of the errors.
- The error characteristics exhibited by different random runs are similar. The exponential convergence behavior of the VarPro errors (before saturation) is quite evident.

Fig. 22 shows the l^∞ (maximum) errors and the l^2 (rms) errors from 10 random runs, together with their statistics, as a function of the number of collocation points obtained with the Gaussian activation function in the neural network. The settings and the simulation parameters here follow those of Fig. 3(b). The results of this figure are in parallel to those of Fig. 21, but for the Gaussian activation function. Note that Fig. 21 is for the cosine activation function. The observations on the VarPro errors from this figure are similar to those from Fig. 21.

Fig. 23 illustrates the l^∞ (maximum) errors and the l^2 (rms) errors of the VarPro solution from 10 random runs, and their statistics, as a function of the number of nodes in the hidden layer, for the cosine activation function in the neural network. The tests, the problem settings and the simulation parameters here follow those of Fig. 4(a). The network architecture is given by $[2, M, 1]$, with M varied systematically. Fig. 23(a) and (b) show the l^∞ and l^2 errors as a function of M from these 10 random runs, respectively, with the random seeds given in the legends. Fig. 23(c) and (d) show the statistical results of the l^∞ error and the l^2 error among these random runs, respectively. The error characteristics exhibited from different random runs are similar. One can observe that the VarPro errors generally decrease exponentially (before saturation) with increasing M in the neural network.

Fig. 24 shows the l^∞ and l^2 errors, and their statistics, from 10 random runs obtained with the Gaussian activation function in the neural network. The settings and the simulation parameters here follow those of Fig. 4(b). The results of this figure for the Gaussian activation function are in parallel to those of Fig. 23 for the cosine activation function. The exponential convergence of the VarPro errors (before saturation) is evident.

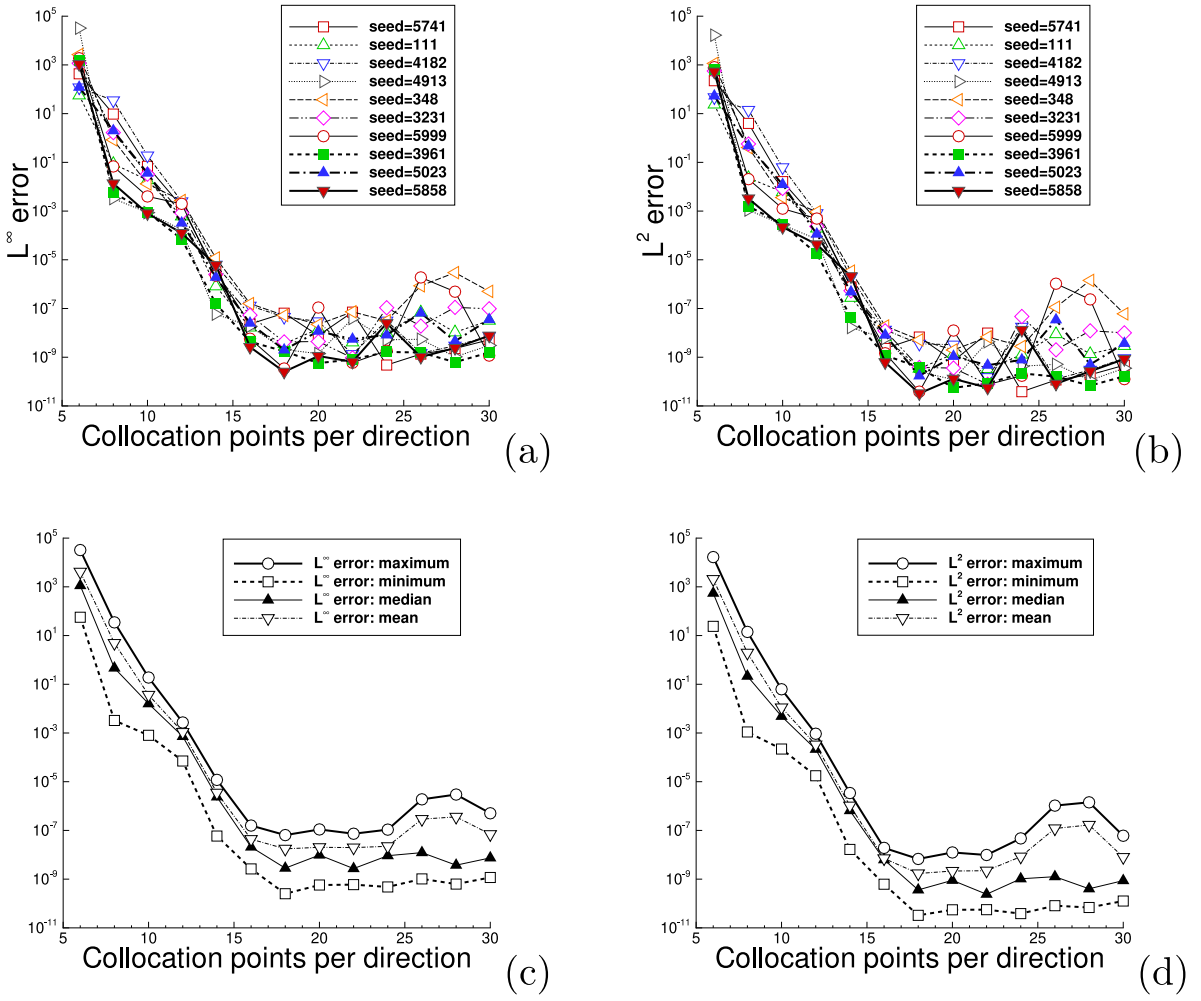

Fig. 21. Appendix A (Poisson equation): The l^∞ (i.e. maximum) error (a) and the l^2 (i.e. rms) error (b) of the VarPro solution versus the number of collocation points per direction, obtained with 10 random seeds in the random number generator (RNG). Here the seed for RNG is set to a random integer generated on [0, 10000), with their specific values given in the legends. The statistics (maximum, minimum, median, and mean) of the 10 random runs for the l^∞ error (c) and for the l^2 error (d) versus the number of collocation points per direction. Network architecture [2, 200, 1], cosine activation function. The settings and simulation parameters here follow those of Fig. 3(a).

Appendix B. Comparison between VarPro and PINN (physics-informed neural network)

This Appendix provides a comparison of the performance between the current VarPro method and the physics-informed neural network (PINN) method from [4] with the Poisson equation (Section 3.1.1) and the nonlinear Helmholtz equation (Section 3.2.1). We refer the reader to [21] for a comparison between the ELM method and the PINN method for several PDEs.

The PINN implementation here follows those in [4,9,21], and is based on the Tensorflow and Keras libraries. In the PINN loss function, we employ a penalty coefficient $\gamma_{bc} \in (0, 1)$ in front of the boundary loss term and a penalty coefficient $(1 - \gamma_{bc})$ in front of the PDE loss term. The coefficient γ_{bc} has been varied systematically for solving the Poisson equation in Section 3.1.1 and the nonlinear Helmholtz equation in Section 3.2.1. We observe that $\gamma_{bc} \approx 0.99$ provides the best results for PINN. The results reported below correspond to $\gamma_{bc} = 0.99$ in the PINN simulations. For a rectangular domain Ω in 2D, we employ a uniform set of $Q = Q_1 \times Q_1$ grid points (with Q_1 points in each direction, on each side of the boundary) as the collocation points in PINN, similar to that in the VarPro simulations. We employ the Adam optimizer [77] for minimizing the loss function in PINN. The learning rate coefficient (i.e. the parameter α in [77]) in Adam is decreased linearly from an initial value to an end value

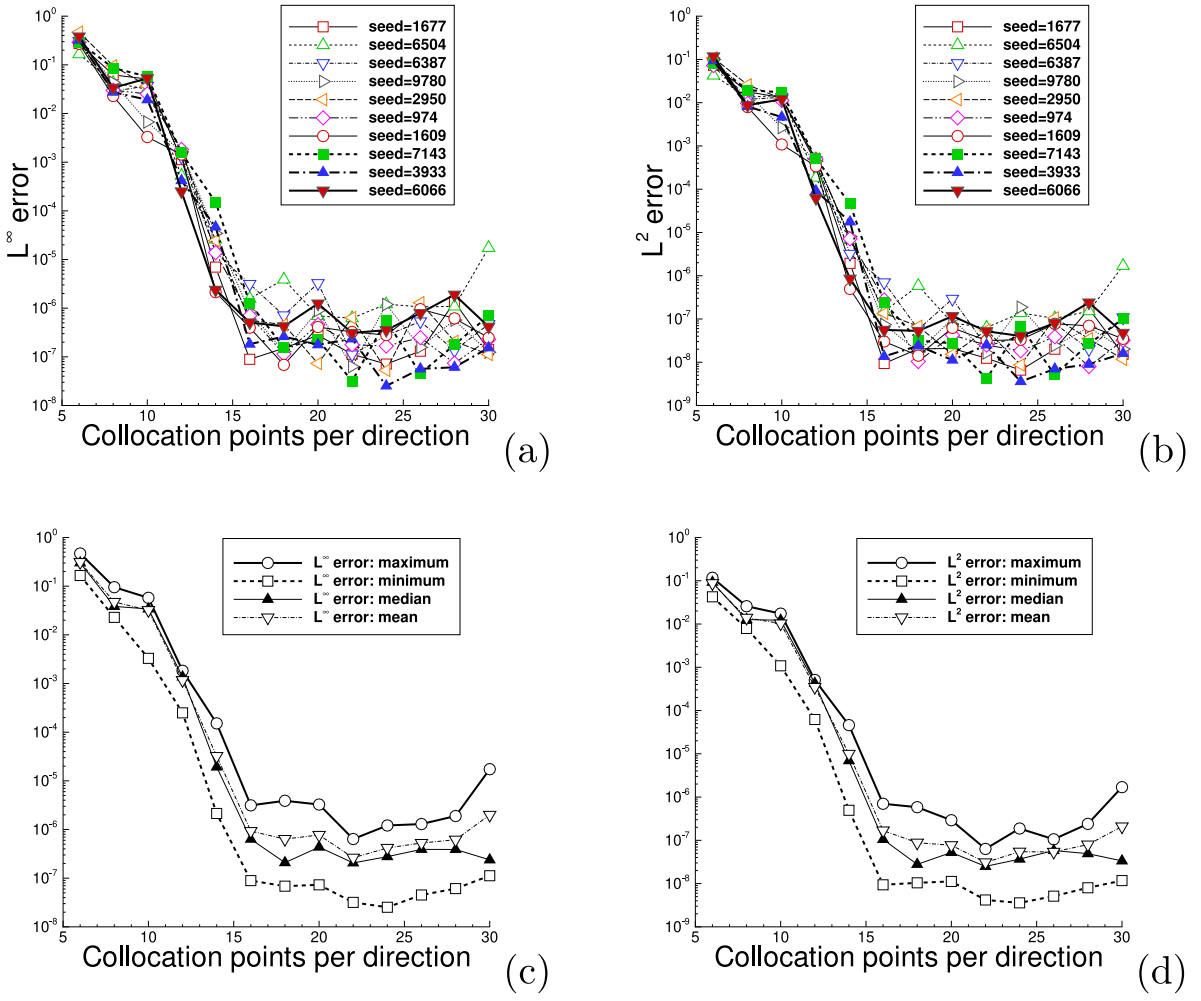

Fig. 22. Appendix A (Poisson equation): The l^∞ (i.e. maximum) error (a) and the l^2 (i.e. rms) error (b) of the VarPro solution versus the number of collocation points per direction, obtained from 10 random runs, with the random seeds given in the legends. The statistics (maximum, minimum, median, and mean) of the 10 random runs for the l^∞ error (c) and for the l^2 error (d) versus the number of collocation points per direction. Network architecture [2, 200, 1], Gaussian activation function. The settings and simulation parameters here follow those of Fig. 3(b).

within a prescribed number of steps (decay steps), and afterwards it is fixed at the end value. We have tried a variety of parameter values in the learning rate schedule, and settled down on the following values for the Poisson equation and the nonlinear Helmholtz equation: initial/end learning rate coefficients $(0.01, 0.0001)$ with decay steps 10,000. For each neural network architecture, we have tried a number of (typically 10) PINN runs with different random initializations. Reported below are the best results among those runs we have obtained with PINN. Upon completion of network training by the Adam optimizer, the PINN network is evaluated on another finer set of $Q_{eval} = Q_2 \times Q_2$ (with $Q_2 = 101$) uniform grid points on the domain to obtain the PINN solution, which is then compared with the exact solution on the same set of points to compute the maximum/rms errors. This is similar to the procedure as discussed in Sections 3.1.1 and 3.2.1 for the VarPro method.

Table 12 compares the maximum/rms errors and the network training time between the VarPro method and the PINN method for solving the boundary value problem (24) with the Poisson equation of Section 3.1.1. We have considered two network architectures given by $[2, 100, 1]$ and $[2, 200, 1]$, respectively, with the "cos" activation function for all hidden nodes and a sequence of uniform collocation points ranging from $Q = 5 \times 5$ to $Q = 30 \times 30$ for network training. The error data for VarPro here correspond to those in Table 2 with $R_m = 1$. The PINN data are

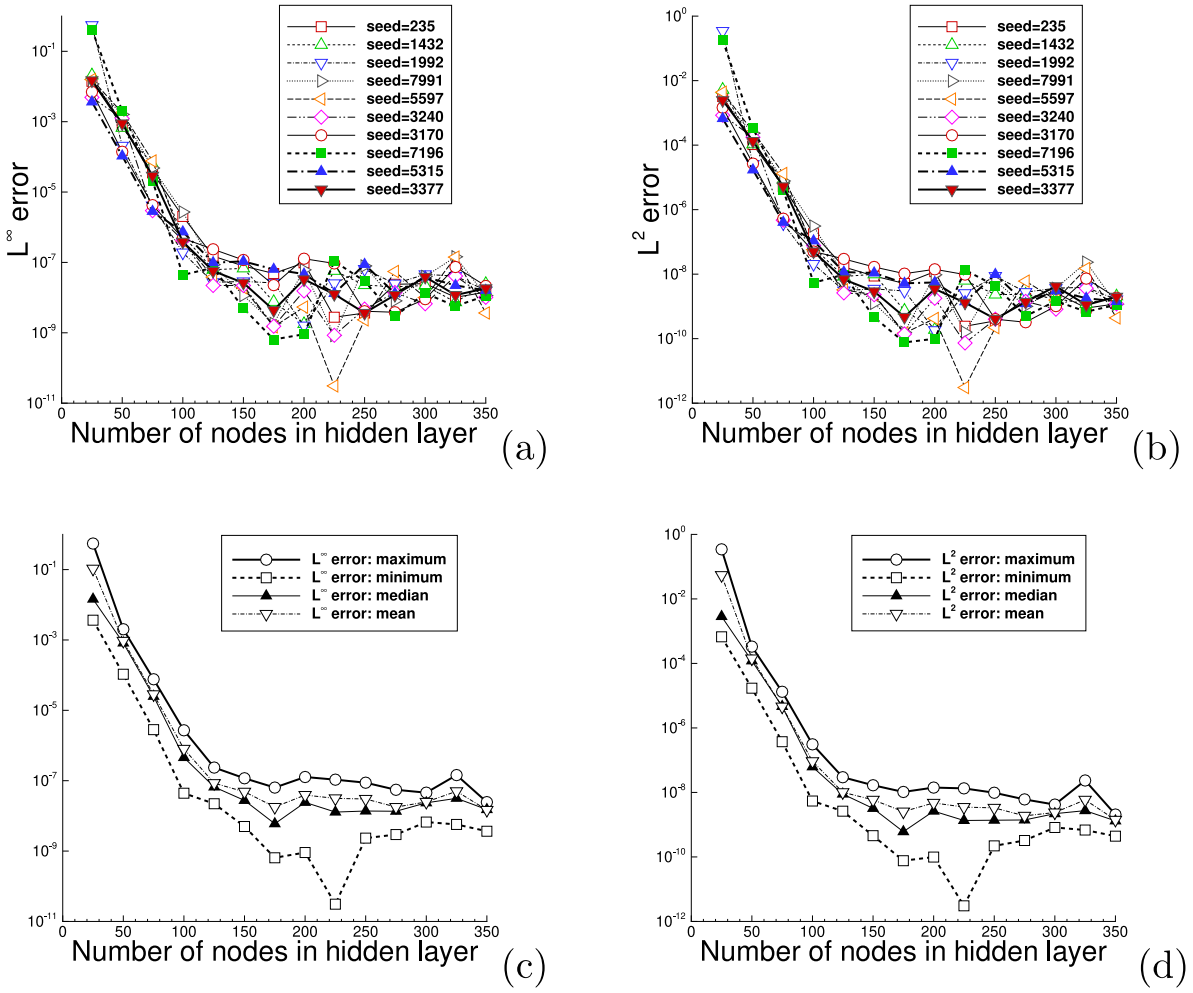

Fig. 23. Appendix A (Poisson equation): The l^∞ (i.e. maximum) error (a) and the l^2 (i.e. rms) error (b) of the VarPro solution versus the number of nodes in the hidden layer (M), obtained from 10 random runs, with the random seeds given in the legends. The statistics (maximum, minimum, median, and mean) of the 10 random runs for the l^∞ error (c) and for the l^2 error (d) versus M. Network architecture $[2, M, 1]$ (M varied), cosine activation function, $Q = 21 \times 21$ uniform collocation points. The settings and simulation parameters here follow those of Fig. 4(a).

obtained with the following parameters. The PINN network is trained for 50,000 epochs with the Adam optimizer. As discussed above, the learning rate coefficient decreases linearly from 0.01 to 10^{-4} in the first 10,000 epochs, and is then fixed at 10^{-4} for the remaining 40,000 epochs.

The data in Table 12 indicate that the VarPro method is considerably more accurate than PINN, with the VarPro errors typically several orders of magnitude smaller (e.g. 10^{-8} versus 10^{-3}), except for the smallest set of collocation points. The network training time of VarPro is also significantly smaller than that of PINN. For VarPro we observe some irregularity in the variation of the network training time with respect to the increase of the collocation points, for example, around 98 s for the 25×25 collocation points versus around 15 s for the 30×30 collocation points with the network $[2, 200, 1]$. This irregularity is caused by the irregularity in the triggering of the subiteration procedure for the initial guess perturbation in Algorithm 3 and the difference in the actual number of subiterations performed. The subiteration procedure is triggered in some cases, but not in others. The network training time will increase notably once it is triggered.

Table 13 provides a comparison of the maximum/rms errors and the network training time of the VarPro and the PINN (Adam optimizer) methods for solving the boundary value problem (28) with the nonlinear Helmholtz

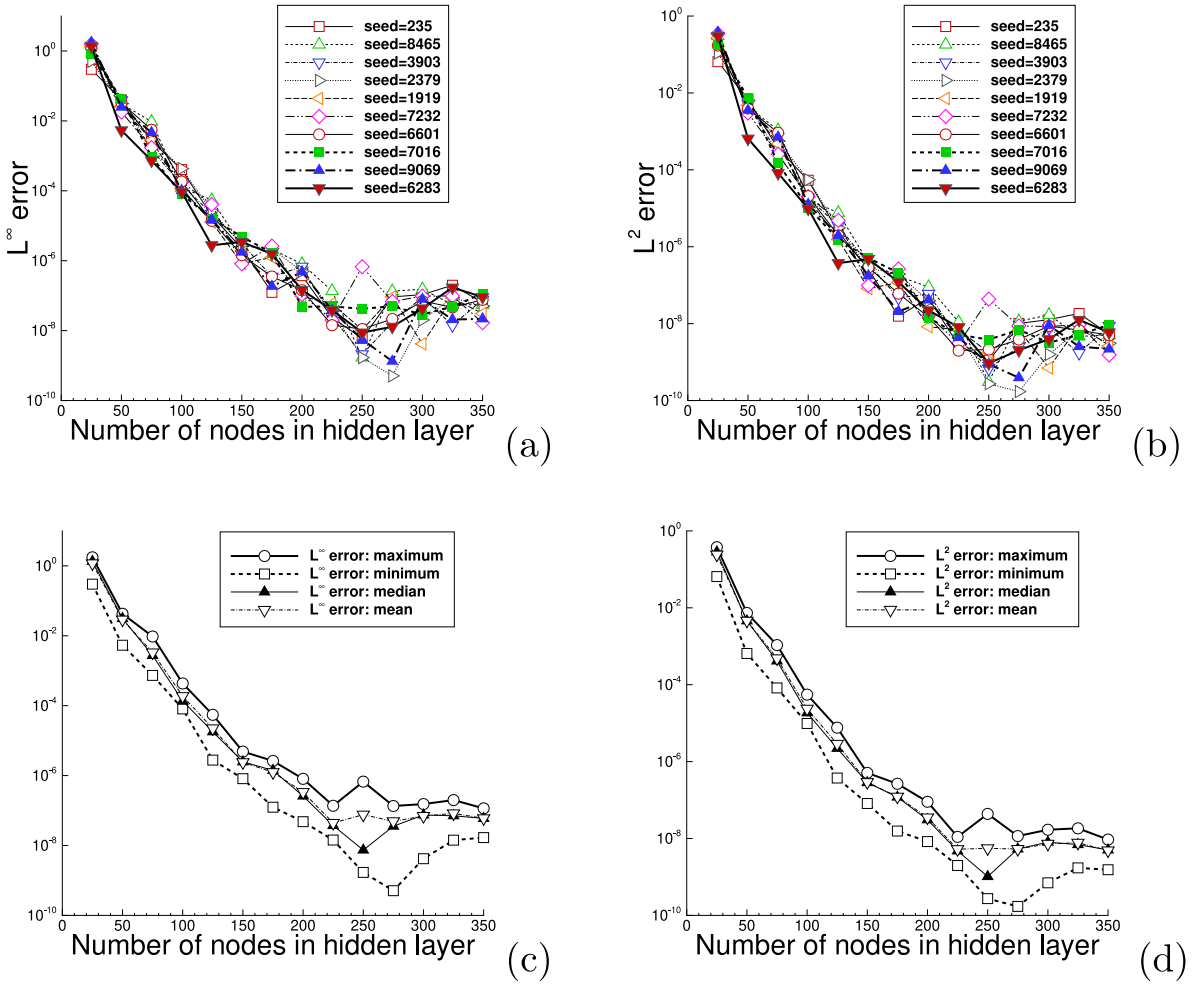

Fig. 24. Appendix A (Poisson equation): The l^∞ (i.e. maximum) error (a) and the l^2 (i.e. rms) error (b) of the VarPro solution versus the number of nodes in the hidden layer (M), obtained from 10 random runs, with the random seeds given in the legends. The statistics (maximum, minimum, median, and mean) of the 10 random runs for the l^∞ error (c) and for the l^2 error (d) versus M. Network architecture $[2, M, 1]$ (M varied), Gaussian activation function, $Q = 21 \times 21$ uniform collocation points. The settings and parameters here follow those of Fig. 4(b).

equation in Section 3.2.1. The network architecture is given by $[2, 200, 1]$ with the sine activation function, and the set of uniform collocation points is varied. The error data for VarPro here correspond to those in Table 6 with $R_m = 1$. In PINN the neural network is trained for 100,000 epochs by the Adam optimizer, with the learning rate coefficient decreasing linearly from 0.01 to 10^{-4} during the first 10,000 epochs and then fixed at 10^{-4} for the remaining iterations. One can again observe that the VarPro errors are in general considerably smaller than the PINN errors (e.g. 10^{-10} versus 10^{-2}), and that the VarPro training time is also much smaller than the PINN training time.

Appendix C. Comparison between VarPro and finite element method

In this appendix we provide a comparison between the VarPro method herein and the classical finite element method (FEM, second-order, linear elements) for solving the Poisson equation (Section 3.1.1) and the nonlinear Helmholtz equation (Section 3.2.1).

The FEM implementation employed here follows those in [21,25]. It is based on Python and the FEniCS library (https://fenicsproject.org/). In FEM simulations we employ a uniform $N \times N$ rectangular mesh, which consists of

Table 12
Appendix B (Poisson equation): comparison of the maximum/rms errors and the network training time (in seconds) obtained using VarPro and PINN (with Adam optimizer). "cos" activation function. The VarPro error data here correspond to those in Table 2 with $R_m = 1$. In PINN, the neural network is trained for 50,000 epochs; the learning rate coefficient decreases linearly from 0.01 to 10^{-4} in the first 10,000 epochs, and then is fixed at 10^{-4} afterwards. Reported here is the best result among several PINN runs with different random initializations.

Neural	Collocation	VarPro			PINN		
network	points	max-error	rms-error	Time (s)	max-error	rms-error	Time (s)
[2, 100, 1]	5 × 5	6.470E+1	2.422E+1	1.85E+0	7.65E−1	2.13E−1	3.14E+1
	10 × 10	8.388E−3	3.941E−3	5.13E+0	1.11E−2	2.03E−3	1.12E+2
	15 × 15	6.018E−7	8.241E−8	2.17E+1	1.83E−2	2.57E−3	1.43E+2
	20 × 20	3.693E−7	4.216E−8	2.88E+1	1.80E−2	2.57E−3	1.81E+2
	25 × 25	5.845E−7	8.054E−8	4.64E+1	2.19E−2	2.95E−3	2.46E+2
	30 × 30	2.688E−7	2.867E−8	4.89E+1	1.67E−2	3.25E−3	3.31E+2
[2, 200, 1]	5 × 5	5.948E+0	2.102E+0	1.92E+0	1.27E+0	4.41E−1	3.31E+1
	10 × 10	2.127E−2	4.398E−3	2.37E+0	2.22E−2	2.86E−3	1.40E+2
	15 × 15	6.082E−8	1.983E−8	5.20E+0	2.32E−2	3.15E−3	1.91E+2
	20 × 20	1.459E−9	1.203E−10	2.15E+1	1.13E−2	1.88E−3	2.99E+2
	25 × 25	1.782E−7	7.978E−8	9.80E+1	2.16E−2	3.23E−3	4.15E+2
	30 × 30	3.000E−9	3.420E−10	1.51E+1	1.32E−2	2.19E−3	5.88E+2

Table 13
Appendix B (Nonlinear Helmholtz equation): comparison of the maximum/rms errors and the network training time (in seconds) of VarPro and PINN (with Adam optimizer). Network architecture [2, 200, 1], "sin" activation function. The VarPro error data here correspond to those in Table 6 with $R_m = 1$. In PINN, the neural network is trained for 100,000 epochs; the learning rate coefficient decreases linearly from 0.01 to 10^{-4} in the first 10,000 epochs, and then is fixed at 10^{-4} afterwards. Reported here is the best result among several PINN runs with different random initializations.

Collocation	VarPro			PINN		
points	max-error	rms-error	Time (s)	max-error	rms-error	Time (s)
5 × 5	1.297E+2	4.536E+1	3.49E+0	1.58E+0	5.92E−1	2.01E+2
10 × 10	1.855E−2	3.955E−3	1.75E+0	9.74E−2	1.59E−2	2.58E+2
15 × 15	2.868E−8	3.252E−9	3.65E+1	7.98E−2	1.73E−2	3.57E+2
20 × 20	3.679E−10	3.203E−11	7.87E+1	7.51E−2	1.59E−2	5.58E+2
25 × 25	6.014E−8	3.281E−9	4.89E+1	7.19E−2	1.34E−2	7.93E+2
30 × 30	5.709E−10	4.576E−11	1.78E+2	7.32E−2	1.22E−2	1.11E+3

Table 14
Appendix C (Poisson equation): main simulation parameters for VarPro and FEM.

	Parameter	Value	Parameter	Value
FEM	Mesh	$N \times N$	Elements	Linear
	N	10, 12, 14, ..., 30	Degrees of freedom	N^2
VarPro	Neural network	$[2, M, 1]$	M	$N^2/4$
	Activation function	Gaussian	Random seed	1
	Training points	$Q_1 \times Q_1$	Q_1	$N/2+1$
	Testing points	$Q_2 \times Q_2$	Q_2	101
	max-subiterations	1	Threshold (Algorithm 3)	$1E-12$
	δ (Algorithm 3)	1.0	p (Algorithm 3)	0.5
	Initial guess $\boldsymbol{\theta}_0$	Random values on $[-1, 1]$	Degrees of freedom	$4M$

$2N^2$ linear triangular elements (each rectangle divided along its diagonal into two triangles) with a total degrees of freedom N^2. The number of elements in each direction, N, is varied in the tests.

Table 15
Appendix C (Poisson equation): comparison of the maximum/rms errors of VarPro and FEM, and their computational cost (VarPro network training time, FEM computation time). See Table 14 for the VarPro/FEM simulation parameter values.

Degrees of	VarPro			FEM		
freedom	max-error	rms-error	Time (s)	max-error	rms-error	Time (s)
100	$3.73E{-}1$	$8.00E{-}2$	$3.3E{+}0$	$4.45E{-}1$	$1.72E{-}1$	$6.9E{-}3$
144	$1.93E{-}1$	$5.85E{-}2$	$2.9E{+}0$	$3.00E{-}1$	$1.22E{-}1$	$7.6E{-}3$
196	$1.14E{-}1$	$1.61E{-}2$	$3.1E{+}0$	$2.28E{-}1$	$9.10E{-}2$	$7.6E{-}3$
256	$2.34E{-}2$	$4.30E{-}3$	$3.7E{+}0$	$1.75E{-}1$	$7.05E{-}2$	$7.5E{-}3$
324	$2.79E{-}3$	$9.47E{-}4$	$4.1E{+}0$	$1.38E{-}1$	$5.62E{-}2$	$8.6E{-}3$
400	$3.59E{-}4$	$9.12E{-}5$	$5.2E{+}0$	$1.13E{-}1$	$4.59E{-}2$	$9.4E{-}3$
484	$5.20E{-}5$	$1.74E{-}5$	$5.9E{+}0$	$9.26E{-}2$	$3.81E{-}2$	$1.1E{-}2$
576	$1.18E{-}5$	$3.09E{-}6$	$8.4E{+}0$	$7.86E{-}2$	$3.22E{-}2$	$1.1E{-}2$
676	$6.60E{-}6$	$2.88E{-}6$	$4.0E{+}0$	$6.69E{-}2$	$2.75E{-}2$	$1.2E{-}2$
784	$1.82E{-}7$	$4.88E{-}8$	$4.2E{+}0$	$5.76E{-}2$	$2.38E{-}2$	$1.4E{-}2$
900	$1.80E{-}6$	$1.31E{-}7$	$6.7E{+}0$	$5.04E{-}2$	$2.08E{-}2$	$1.5E{-}2$

Table 16
Appendix C (Nonlinear Helmholtz equation): main simulation parameters for VarPro and FEM.

	Parameter	Value	Parameter	Value
FEM	Mesh	$N \times N$	Elements	Linear
	N	10, 12, 14, ..., 30	Degrees of freedom	N^2
VarPro	Neural network	$[2, M, 1]$	M	$N^2/4$
	Activation function	Gaussian	Random seed	1
	Training points	$Q_1 \times Q_1$	Q_1	$N/2+2$
	Testing points	$Q_2 \times Q_2$	Q_2	101
	max-subiterations	0	Threshold (Algorithm 3)	$1E{-}12$
	δ (Algorithm 3)	Not used	p (Algorithm 3)	Not used
	Initial guess $\boldsymbol{\theta}_0$	Random values on $[-1, 1]$	Degrees of freedom	$4M$
	max-iterations-newton	15	Tolerance-newton	$1E{-}12$

In VarPro simulations we employ a neural network architecture $[2, M, 1]$, with $M = N^2/4$. Since the total number of training parameters in the neural network is $4M$, this setting results in the same total number of degrees of freedom in the VarPro and FEM simulations. The Gaussian activation function is employed for all the hidden nodes. For network training we employ a uniform set of $Q = Q_1 \times Q_1$ collocation points, where we set $Q_1 = N/2+1$ for the linear problem and $Q_1 = N/2+2$ for the nonlinear problem. We vary the degrees of freedom in the VarPro/FEM simulations by varying N systematically between 10 and 30. The maximum/rms errors of the VarPro/FEM results, as well as the VarPro network training time and the FEM computation time, are recorded for comparisons.

In Tables 14 and 15 we summarize the VarPro/FEM results for solving the boundary value problem (24) with the Poisson equation. Table 14 lists the parameter values in the FEM and VarPro simulations for this problem. Table 15 is a comparison of the maximum/rms errors between VarPro and FEM, and their computational cost (VarPro network training time, FEM computation time), versus the number of degrees of freedom in the system. It can be observed that, under the same degrees of freedom, the VarPro method is considerably more accurate than the FEM, and its computational cost is also much higher than the latter.

Tables 16 and 17 provide a summary of the VarPro/FEM results for solving the boundary value problem (28) with the nonlinear Helmholtz equation. Table 16 lists the parameter values in the VarPro and FEM simulations for this problem. Table 17 is a comparison of the maximum/rms errors between VarPro and FEM, as well as their computational cost (VarPro network training time, FEM computation time), for the nonlinear Helmholtz equation. We arrive at the same conclusion as with the Poisson equation. The VarPro method is much more accurate than FEM, but its computational cost is also much higher than that of FEM, under the same degrees of freedom in the system.

Table 17
Appendix C (Nonlinear Helmholtz equation): comparison of the maximum/rms errors of VarPro and FEM, and their computational cost (VarPro network training time, FEM computation time). See Table 16 for the VarPro/FEM simulation parameter values.

Degrees of	VarPro			FEM		
freedom	max-error	rms-error	Time (s)	max-error	rms-error	Time (s)
100	$1.61E-1$	$3.39E-2$	$5.5E+0$	$1.76E-1$	$7.41E-2$	$2.0E-2$
144	$1.07E-1$	$1.46E-2$	$6.5E+0$	$1.25E-1$	$5.19E-2$	$2.4E-2$
196	$2.06E-2$	$5.49E-3$	$8.7E+0$	$9.10E-2$	$3.85E-2$	$2.8E-2$
256	$3.73E-3$	$9.75E-4$	$1.32E+1$	$6.78E-2$	$2.97E-2$	$2.9E-2$
324	$7.31E-4$	$1.44E-4$	$1.53E+1$	$5.56E-2$	$2.36E-2$	$3.4E-2$
400	$1.26E-4$	$2.42E-5$	$2.08E+1$	$4.52E-2$	$1.92E-2$	$4.0E-2$
484	$1.40E-4$	$2.88E-5$	$1.92E+1$	$3.70E-2$	$1.60E-2$	$4.5E-2$
576	$3.84E-5$	$4.38E-6$	$2.22E+1$	$3.14E-2$	$1.35E-2$	$5.2E-2$
676	$4.82E-6$	$9.37E-7$	$2.59E+1$	$2.69E-2$	$1.15E-2$	$5.8E-2$
784	$7.56E-7$	$1.08E-7$	$2.05E+1$	$2.30E-2$	$9.95E-3$	$6.8E-2$
900	$2.25E-6$	$3.76E-7$	$2.20E+1$	$2.00E-2$	$8.69E-3$	$7.8E-2$

References

[1] G.H. Golub, V. Pereyra, The differentiation of pseudo-inverse and nonlinear least squares problems whose variables separate, SIAM J. Numer. Anal. 10 (1973) 413–432.
[2] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, The MIT Press, 2016.
[3] J. Sirignano, K. Spoliopoulos, DGM: A deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018) 1339–1364.
[4] 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.
[5] W. E, B. Yu, The deep Ritz method: a deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (2018) 1–12.
[6] J. He, J. Xu, MgNet: A unified framework for multigrid and convolutional neural network, Sci. China Math. 62 (2019) 1331–1354.
[7] T. Luo, H. Yang, Two-layer neural networks for partial differential equations: optimization and generlization theory, 2020, arXiv:2006.15733.
[8] Y. Zang, G. Bao, X. Ye, H. Zhou, Weak adversarial networks for high-dimensional partial differential equations, J. Comput. Phys. 411 (2020) 109409.
[9] S. Dong, N. Ni, A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks, J. Comput. Phys. 435 (2021) 110242.
[10] E. Samanaiego, C. Anitescu, S. Goswami, V.M. Nguyen-Thanh, H. Guo, K. Hamdia, X. Zhuang, T. Rabczuk, An energy approach to the solution of partial differential equations in computational mechanics via machine learning: concepts, implementation and applications, Comput. Methods Appl. Mech. Engrg. 362 (2020) 112790.
[11] S. Wang, X. Yu, P. Perdikaris, When and why PINNs fail to train: a neural tangent kernel perspective, J. Comput. Phys. 449 (2022) 110768.
[12] Z. Mao, A.D. Jagtap, G.E. Karniadakis, Physics-informed neural networks for high-speed flows, Comput. Methods Appl. Mech. Engrg. 360 (2020) 112789.
[13] H. You, Y. Yu, N. Trask, M. Gulian, M. D'Elia, Data-driven learning of nonlocal physics from high-fidelity synthetic data, Comput. Methods Appl. Mech. Engrg. 374 (2021) 113553.
[14] A.S. Krishnapriyan, A. Gholami, S. Zhe, R.M. Kirby, M.W. Mahoney, Characterizing possible failure modes in physics-informed neural networks, 2021, arXiv:2109.01050.
[15] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, DeepXDE: A deep learning library for solving differential equations, SIAM Rev. 63 (2021) 208–228.
[16] S. Liang, S.W. Jiang, J. Harlim, H. Yang, Solving PDEs on unknown manifolds with machine learning, 2021, arXiv:2106.06682.
[17] M. Penwarden, S. Zhe, A. Narayan, R.M. Kirby, Physics-informed neural networks (pinns) for parameterized PDEs: a metalearning approach, 2021, arXiv:2110.13361.
[18] G.E. Karniadakis, G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Physics-informed machine learning, Nat. Rev. Phys. 3 (2021) 422–440.
[19] W. Hao, X. Jin, J.W. Siegel, J. Xu, An efficient greedy algorithm for neural networks and applications in PDEs, 2021, arXiv:2107.04466.
[20] V. Dwivedi, B. Srinivasan, Physics informed extreme learning machine (pielm) - a rapid method for the numerical solution of partial differential equations, Neurocomputing 391 (2020) 96–118.
[21] S. Dong, Z. Li, Local extreme learning machines and domain decomposition for solving linear and nonlinear partial differential equations, Comput. Methods Appl. Mech. Engrg. 387 (2021) 114129, (also arXiv:2012.02895).

[22] S. Dong, Z. Li, A modified batch intrinsic plascity method for pre-training the random coefficients of extreme learning machines, J. Comput. Phys. 445 (2021) 110585, (also arXiv:2103.08042).
[23] F. Calabro, G. Fabiani, C. Siettos, Extreme learning machine collocation for the numerical solution of elliptic PDEs with sharp gradients, Comput. Methods Appl. Mech. Engrg. 387 (2021) 114188.
[24] G. Fabiani, F. Calabro, L. Russo, C. Siettos, Numerical solution and bifurcation analysis of nonlinear partial differential equations with extreme learning machines, J. Sci. Comput. 89 (2021) 44.
[25] S. Dong, J. Yang, On computing the hyperparameter of extreme learning machines: algorithm and application to computational PDEs and comparison with classical and high-order finite elements, J. Comput. Phys. 463 (2022) 111290, (also arXiv:2110.14121).
[26] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (2006) 489–501.
[27] G.B. Huang, L. Chen, C.-K. Siew, Universal approximation using incremental constructive feedforward networks with random hidden nodes, IEEE Trans. Neural Netw. 17 (2006) 879–892.
[28] G.H. Golub, V. Pereyra, Separable nonlinear least squares: the variable projection method and its applications, Inverse Problems 19 (2003) R1–R26.
[29] J.E. Dennis, R.B. Schnabel, Numerical Methods for Unconstrained Optimization and Nonlinear Equations, SIAM, 1996.
[30] A. Bjorck, Numerical Methods in Matrix Computations, Springer, 2015.
[31] A. Ruhe, P.A. Wedin, Algorithms for separable nonlinear least squares problems, SIAM Rev. 22 (1980) 318–337.
[32] J. Sjoberg, M. Viberg, Separable nonlinear least squares minimization - possible improvements for neural net fitting, in: Neural Networks for Signal Processing VII. Proceedings of IEEE Signal Processing Workshop, 1997.
[33] G.E. Karniadakis, S.J. Sherwin, Spectral/Hp Element Methods for Computational Fluid Dynamics, second ed., Oxford University Press, 2005.
[34] L. Kaufman, A variable projection method for solving separable nonlinear least squares problems, BIT 15 (1975) 49–57.
[35] J. Chung, E. Haber, J. Nagy, Numerical methods for coupled super-resolution, Inverse Problems 22 (2006) 1261–1272.
[36] M.R. Osborne, Separable least squares, variable projection, and the gauss-newton algorithm, Electron. Trans. Numer. Anal. 28 (2007) 1–15.
[37] K.M. Mullen, I.H.M. van Stokkum, The variable projection algorithm in time-resolved spectroscopy, microscopy and mass spectrometry applications, Numer. Algorithms 51 (2009) 319–340.
[38] D.P. O'Leary, B.W. Rust, Variable projection for nonlinear least squares problems, Comput. Optim. Appl. 54 (2013) 579–593.
[39] T. Askham, J.N. Kutz, Variable projection methods for an optimized dynamic mode decomposition, SIAM J. Appl. Dyn. Syst. 17 (2018) 380–416.
[40] G.-Y. Chen, M. Gan, C.L.P. Chen, H.-X. Li, A regularized variable projection algorithm for separable nonlinear least-squares problems, IEEE Trans. Automat. Control 64 (2019) 526–537.
[41] X. Song, W. Xu, K. Hayami, N. Zheng, Secant variable projection method for solving nonnegative separable least squares problems, Numer. Algorithms 85 (2020) 737–761.
[42] N.B. Erichson, P. Zheng, K. Manohar, J.N. Kutz, S.L. Brunton, A.Y. Aravkin, Sparse principal component analysis via variable projection, SIAM J. Appl. Math. 80 (2020) 977–1002.
[43] T. van Leeuwen, A.Y. Aravkin, Variable projection for nonsmooth problems, SIAM J. Sci. Comput. 43 (2021) S249–S268.
[44] E. Newman, J. Chung, M. Chung, L. Ruthotto, SlimTrain – a stochastic approximation method for training separable deep neural networks, 2021, arXiv:2109.14002.
[45] M. Gan, C.L.P. Chen, H.-Y. Chen, L. Chen, On some separated algorithms for separable nonlinear least squares problems, IEEE Trans. Cybern. 48 (2018) 2866–2974.
[46] L. Kaufman, V. Pereyra, A method for separable nonlinear least squares problems with separable equality constraints, SIAM J. Numer. Anal. 15 (1978) 12–20.
[47] D.M. Sima, S. Van Huffel, Separable nonlinear least squares fitting with linear bound constraints and its application in magnetic resonance spectroscopy data quantification, J. Comput. Appl. Math. 203 (2007) 264–278.
[48] A. Cornelio, E.L. Piccolomini, J.G. Nagy, Constrained numerical optimization methods for blind deconvolution, Numer. Algorithms 65 (2014) 23–42.
[49] F.T. Krogh, Efficient implementation of a variable projection algorithm for nonlinear least squares problems, Commun. ACM 17 (1974) 167–169.
[50] A.Y. Aravkin, T. van Leeuwen, Estimating nuisance parameters in inverse problems, Inverse Problems 28 (2012) 115016.
[51] P. Shearer, A.C. Gilbert, A generalization of variable elimination for separable inverse problems beyond least squares, Inverse Problems 29 (2013) 045003.
[52] J.L. Herring, J.G. Nagy, L. Ruthotto, LAP: A linearize and project method for solving inverse problems with coupled variables, Sampl. Theory Signal Image Process. 17 (2018) 127–151.
[53] A.E.B. Ruano, D.J. Jones, P.J. Fleming, A new formulation of the learning problem of a neural network controller, in: Proc. 30th IEEE Conf. Decis. Control, Brighton, UK, 1991, pp. 865–866.
[54] S. McLoone, M.D. Brown, G. Irwin, A hybrid linear/nonlinear training algorithm for feedforward neural networks, IEEE Trans. Neural Netw. 9 (1998) 669–684.
[55] J. Nocedal, S. Wright, Numerical Optimization, Springer, 1999.
[56] E.C. Cyr, M.A. Gulian, R.G. Patel, M. Perego, N.A. Trask, Robust training and initialization of deep neural networks: an adaptive basis viewpoint, Proc. Mach. Learn. Res. 107 (2020) 1–26.
[57] K. Weigl, M. Berthod, Neural Networks as Dynamical Bases in Function Space, Report No 2124, INRIA, Sophis-Antipolis, France, 1993, URL: https://hal.inria.fr/inria-00074548/document.

[58] K. Weigl, G. Giraudon, M. Berthod, Application of Projection Learning to the Detection of Urban Areas in SPOT Satellite Images, Report No 2143, INRIA, Sophia-Antipolis, France, 1993, URL: https://hal.inria.fr/inria-00074529.
[59] K. Weigl, M. Berthod, Projection learning: alternative approach to the computation of the projection, in: Proc. European Symp. on Artificial Neural Networks, Brussels, Belgium, 1994, pp. 19–24.
[60] V. Pereyra, G. Scherer, F. Wong, Variable projections neural network training, Math. Comput. Simulation 73 (2006) 231–243.
[61] C.-T. Kim, J.-J. Lee, Training two-layered feedforward networks with variable projection method, IEEE Trans. Neural Netw. 19 (2008) 371–375.
[62] E. Newman, L. Ruthotto, J. Hart, B. van Bloemen Waanders, Train like a (Var)Pro: Efficient training of neural networks with variable projection, 2020, arXiv:2007.13171.
[63] B. Szabo, I. Babushka, Finite Element Analysis, John Wiley & Sons, Inc., 1991.
[64] Y. Yu, R.M. Kirby, G.E. Karniadakis, Spectral element and hp methods, in: Encyclopedia of Computational Mechanics, Vol. 1, John Wiley and Sons, NY, 2017, pp. 1–43.
[65] X. Zheng, S. Dong, An eigen-based high-order expansion basis for structured spectral elements, J. Comput. Phys. 230 (2011) 8573–8602.
[66] S. Dong, J. Shen, A time-stepping scheme involving constant coefficient matrices for phase field simulations of two-phase incompressible flows with large density ratios, J. Comput. Phys. 231 (2012) 5788–5804.
[67] S. Dong, Multiphase flows of N immiscible incompressible fluids: a reduction-consistent and thermodynamically-consistent formulation and associated algorithm, J. Comput. Phys. 361 (2018) 1–49.
[68] S. Dong, A convective-like energy-stable open boundary condition for simulations of incompressible flows, J. Comput. Phys. 302 (2015) 300–328.
[69] L. Lin, Z. Yang, S. Dong, Numerical approximation of incompressible Navier-Stokes equations based on an auxiliary energy variable, J. Comput. Phys. 388 (2019) 1–22.
[70] Z. Yang, S. Dong, An unconditionally energy-stable scheme based on an implicit auxiliary energy variable for incompressible two-phase flows with different densities involving only precomputable coefficient matrices, J. Comput. Phys. 393 (2019) 229–257.
[71] Z. Yang, S. Dong, A roadmap for discretely energy-stable schemes for dissipative systems based on a generalized auxiliary variable with guaranteed positivity, J. Comput. Phys. 404 (2020) 109121, (also arXiv:1904.00141).
[72] M.A. Branch, T.F. Coleman, Y. Li, A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems, SIAM J. Sci. Comput. 21 (1999) 1–23.
[73] R.H. Byrd, R.B. Schnabel, G.A. Shultz, Approximate solution of the trust region problem by minimization over two-dimensional subspaces, Math. Program. 40 (1988) 247–263.
[74] D. Hendrycks, K. Gimpel, Gaussian error linear units (GELU), 2016, arXiv:1606.08415.
[75] C. Basdevant, M. Deville, P. Haldenwang, J.M. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, A.T. Patera, Spectral and finite difference solutions of the Burgers equation, Comput. & Fluids 14 (1986) 23–41.
[76] W. Strauss, Numerical solution of nonlinear klein-gordon equation, J. Comput. Phys. 28 (1978) 271–278.
[77] D.P. Kingma, J. Ba, Adam: a method for stochastic optimization, 2014, arXiv:1412.6980.