Available online at www.sciencedirect.com
ScienceDirect
Comput. Methods Appl. Mech. Engrg. 393 (2022) 114710
Computer methods in applied mechanics and engineering
www.elsevier.com/locate/cma

On quadrature rules for solving Partial Differential Equations using Neural Networks

Jon A. Rivera[a,b,*], Jamie M. Taylor[b], Ángel J. Omella[a], David Pardo[a,b,c]

[a] University of the Basque Country (UPV/EHU), Leioa, Spain
[b] Basque Center for Applied Mathematics (BCAM), Bilbao, Spain
[c] Ikerbasque (Basque Foundation for Sciences), Bilbao, Spain

Received 31 October 2021; received in revised form 25 January 2022; accepted 30 January 2022
Available online 23 February 2022

Abstract

Neural Networks have been widely used to solve Partial Differential Equations. These methods require to approximate definite integrals using quadrature rules. Here, we illustrate via 1D numerical examples the quadrature problems that may arise in these applications and propose several alternatives to overcome them, namely: Monte Carlo methods, adaptive integration, polynomial approximations of the Neural Network output, and the inclusion of regularization terms in the loss. We also discuss the advantages and limitations of each proposed numerical integration scheme. We advocate the use of Monte Carlo methods for high dimensions (above 3 or 4), and adaptive integration or polynomial approximations for low dimensions (3 or below). The use of regularization terms is a mathematically elegant alternative that is valid for any spatial dimension; however, it requires certain regularity assumptions on the solution and complex mathematical analysis when dealing with sophisticated Neural Networks.

MSC: 35A25; 65N50; 68T07

Keywords: Deep learning; Neural Networks; Ritz method; Least-Squares method; Quadrature rules

1. Introduction

In the last years, the use of Deep Learning (DL) methods has grown exponentially in multiple areas, including self-driving cars [1,2], speech recognition [3,4], and healthcare [5,6]. Similarly, the use of DL algorithms has become popular for solving Partial Differential Equations (PDEs) — see, e.g., [7–14].

DL techniques present several advantages and limitations with respect to traditional PDE solvers based on Finite Elements (FE) [15], Finite Differences (FD) [16], or Isogeometric Analysis (IGA) [17]. Among the advantages of DL, we encounter the nonnecessity of generating a grid. In general, DL uses a dataset in which each datum is independent from others. In contrast, in the linear system that produces the FEM method there exists a connectivity between the nodes of the mesh. The independence of the data in DL allows the parallelization into GPUs for fast computations. Furthermore, DL provides the possibility of solving certain problems that cannot be solved via

* Corresponding author at: University of the Basque Country (UPV/EHU), Leioa, Spain.
E-mail address: jonander.rivera@ehu.eus (J.A. Rivera).

https://doi.org/10.1016/j.cma.2022.114710

traditional methods, like high-dimensional PDEs [18,19], some fractional PDEs [20,21], and multiple nonlinear PDEs [22,23].

However, DL also presents limitations when solving PDEs. For example, in [24] they show that Fully-Connected Neural Networks suffer from spectral bias [25] and exhibit different convergence rates for each loss component. In addition, the convergence of the method is often assumed (see, e.g., [26]) since it cannot be rigorously guaranteed due to the non-convexity of the loss function. Another notorious problem is due to quadrature errors. In traditional mesh-based methods, such as FEM, we first select an approximating solution space, and then compute the necessary integrals over each element required to produce the stiffness matrix. With DL, we first set a quadrature rule and then construct the approximated function. Due to this, there is no proper quadrature rule for Deep Neural Networks, as remarked in [27]. As we will show throughout this work, quadrature errors may be extremely large due to the unknown form of the integrand. These errors can also be interpreted as a form of overfitting [28,29] that applies to the PDE constraint (e.g., $u" = 0$) rather than to the PDE solution (e.g., u). This may have disastrous consequences since it may result in approximate solutions that are far away from the exact ones at all points.

There exist different methods to approximate definite integrals by discrete data. The most common ones used in DL are based on Monte Carlo integration. These methods compute definite integrals using randomly sampled points from a given distribution. Monte Carlo integration is suitable for high-dimensional integrals. However, for low-dimensional integrals (1D, 2D, and 3D), convergence is slow in terms of the number of integration points in comparison to other quadrature rules. This produces elevated computational costs. Examples of existing works that follow a Monte Carlo approach are [30] using a Deep Ritz Method (DRM), [31] using a Deep Galerkin Method (DGM), and [32], where they employ the so-called PINNs [33] that can be interpreted as a collocation method or as a variational method using Dirac delta test functions. From the practical point of view, at each iteration of the Stochastic Gradient Descent (SGD), they consider a mini-batch of randomly selected points to discretize the domain.

Another existing method to compute integrals in DNNs is the so-called *automatic integration* [34]. In this method, the author approximates the integrand by its high order Taylor series expansion around a given point within the integration domain. Then, the integrals are computed analytically. The derivatives needed in the Taylor series expansion are computed via automatic differentiation (a.k.a *autodiff*) [35]. Since the information of the derivatives is local, overfitting may easily occur.

Another quadrature rule alternative is to use adaptive integration methods. Examples where authors have selected these methods are the Deep Least Square (DLS) method [36], where they use an adaptive mid-point quadrature rule using local error indicators. These indicators are based on the value of the residual at randomly selected points, and the resulting quadrature error is unclear.

In Variational Neural Networks (VarNets) [37], the authors employ a Gauss quadrature rule to evaluate their integrals. As error indicators, they use the strong form of the residual evaluated over a set of random points that follow a uniform distribution.

One can also use Gauss-type quadrature rules to evaluate integrals, as they do in Variational Physics-Informed Neural Networks (VPINNs) [27]. One limitation of this method is the impossibility of selecting *a priori* an adequate quadrature order because properties of the Neural Network approximation are unknown. In addition, the use of fixed quadrature rules increase the chances of performing overfitting.

While the choice of an adequate quadrature rule is critical for Deep Learning solution of PDEs, and while different numerical integration schemes exist for this purpose, a work dedicated to analyzing existing quadrature rule alternatives for DNNs is missing in the literature. Herein, we analyze the problems associated with quadrature rules in DL methods when solving PDEs, and we propose several alternatives to overcome the quadrature problems: Monte Carlo integration, exact (Gaussian) integration of piecewise-polynomial approximations, adaptive integration, and the use of regularizers. We discuss their advantages and limitations and develop adequate regularizers for certain problems using similar ideas to those presented in [26]. Moreover, we illustrate with numerical examples the different approximated solutions obtained with the different integration strategies.

For illustration purposes, we focus on a simple one-dimensional (1D) Laplace problem and the Deep Ritz Method (DRM) [30] as well as the Deep Least-Squares method (DLS) [36]. However, the work presented here and drawn conclusions are extendable to (a) higher spatial dimensions, (b) different PDEs, and (c) most existing DNN solution methods, including PINNs [33]. In the same way, the integration methods presented in this work are valid for different geometries and boundary conditions from the ones considered in this work.

The remainder of this article is as follows. Section 2 presents our selected model problem. Section 3 defines the loss functions used in this work for solving PDEs. Section 4 describes our Neural Network (NN) and explains some critical implementation aspects. Section 5 illustrates the quadrature problems via two numerical examples. Section 6 proposes several methods to overcome the quadrature problems and Section 7 shows numerical results corresponding to the proposed integration strategies. Finally, Section 8 summarizes the main findings and possible future lines of research in the area.

2. Model problem

Let $\Omega \subset \mathbb{R}$ be a computational domain, and Γ_D and Γ_N two disjoint sections of its boundary, where $\Gamma_D \cup \Gamma_N = \partial\Omega$ and the subscripts D and N denote the Dirichlet and Neumann bounds, respectively. We consider the following boundary value problem:

$$\begin{cases} -u'' = f & x \in \Omega, \quad \text{(a)} \\ u = 0 & x \in \Gamma_D, \quad \text{(b)} \\ u' \cdot \mathbf{n} = g & x \in \Gamma_N. \quad \text{(c)} \end{cases} \tag{1a}$$

In the above, $\mathbf{n}$ is the unit normal outward (to the domain) vector, and we assume the usual regularity assumptions, namely, $f \in L^2(\Omega)$, $g \in H^{-1/2}(\Gamma_N)$, and $u \in V = H_0^1(\Omega) = \{v \in H^1(\Omega) \text{ and } v|_{\Gamma_D} = 0\}$, where $H^1(\Omega) = \{v \in L^2(\Omega), v' \in L^2(\Omega)\}$.

In this work we solve two different model problems to illustrate our numerical results.

2.1. Model problem 1

The solution of model problem 1 is $u(x) = x^{0.7}$ and it satisfies Eq. (2).

$$\begin{cases} -u''(x) = 0.21x^{-1.3} & x \in (0, 10), \\ u(0) = 0, \\ u'(10) = \frac{0.7}{10^{0.3}}. \end{cases} \tag{2}$$

2.2. Model problem 2

The solution of model problem 2 is $u(x) = x^2$ and it satisfies Eq. (3).

$$\begin{cases} u''(x) = 2 & x \in (0, 10), \\ u(0) = 0, \\ u'(10) = 20. \end{cases} \tag{3}$$

3. Loss functions

We introduce the following standard $L^2(\Omega)$ inner products:

$$(u, v) = \int_\Omega u\, v \qquad \text{and} \qquad (g, v)_{\Gamma_N} = \int_{\Gamma_N} g\, v. \tag{4}$$

In the following, we consider two methods: the Ritz Method [38], and the Least Squares Method [39].

3.1. Ritz method

Multiplying the PDE from Eq. (1a) by a test function $v \in V$ (where $V = H_0^1(\Omega)$), integrating by parts and incorporating the boundary conditions, we arrive at the variational formulation:

$$\text{Find } u \in V \text{ such that } (u', v') = (f, v) + (g, v)_{\Gamma_N} \qquad \forall v \in V. \tag{5}$$

It is easy to prove that the original (strong) and variational (weak) formulations are equivalent (see, for instance, [40]).

To introduce the Ritz method, we define the energy function $\mathcal{F}_R : V \longrightarrow \mathbb{R}$ given by

$$\mathcal{F}_R(v) = \frac{1}{2}(v', v') - (f, v) - (g, v)_{\Gamma_N}. \tag{6}$$

As proved in [40], problem (5) is equivalent to the following energy minimization problem:

$$u = \underset{v \in V}{\arg\min}\ \mathcal{F}_R(v), \tag{7}$$

3.2. Least Squares (LS) method

Reordering the terms of Eq. (1a) and (1a), we define:

$$\mathcal{G}u := u'' + f \qquad x \in \Omega, \tag{8a}$$

$$\mathcal{B}u := u' \cdot \mathbf{n} - g \qquad x \in \Gamma_N. \tag{8b}$$

To introduce the Least Squares method, we define the function $\mathcal{F}_{LS} : V \longrightarrow \mathbb{R}$, where the function $v \in V$ satisfies the Dirichlet conditions:

$$\mathcal{F}_{LS}(v) = |(\mathcal{G}v, \mathcal{G}v)| + |(\mathcal{B}v, \mathcal{B}v)_{\Gamma_N}|. \tag{9}$$

We want to minimize the function $\mathcal{F}_{LS}(v)$ subject to the essential (Dirichlet) BCs. We often find the minimum by taking the derivative equal to zero and ending up with a linear system of equations. In the context of DL, we can simply introduce the above loss function $\mathcal{F}_{LS}(v)$ directly in our NN. Therefore, we want to find

$$u = \underset{v \in V}{\arg\min}\ \mathcal{F}_{LS}(v). \tag{10}$$

4. Neural network implementation

We train a Neural Network, named $u_{NN}(x; \theta)$, with the following architecture. In this first part, we define the trainable part of our NN, with learnable parameters θ. We call it u_θ. It is composed of:

(1) An input layer. This layer receives the data in the form of a $n \times d$ vector, where n is the number of samples and d is the dimension of the data of the problem.
(2) One hidden dense layer with n neurons and a *sigmoid* activation function.
(3) An output layer that delivers u_θ.

Now, we add more layers to our scheme in order to implement Eqs. (6) or (9). For that, we introduce:

(4) A non-trainable layer to impose the Dirichlet boundary conditions. For that, we select a function $\phi(x)$ that satisfies the Dirichlet conditions of the problem and its value is nonzero everywhere else [41]. In this work, we select the following $\phi(x)$ functions for 1D problems in the interval $\Omega = [a, b]$:

$$\phi(x) = \prod_{x_D \in \Gamma_D} (x - x_D). \tag{11}$$

Then, we generate a new output of the NN: $u_{NN}(x; \theta) = \phi(x) u_\theta(x)$ that strongly imposes the homogeneous Dirichlet boundary conditions.

(5) A non-trainable layer to compute the loss function $\mathcal{F}_R$ or $\mathcal{F}_{LS}$ following Eqs. (6) or (9). Within this layer, we evaluate the integrals and the derivatives. We consider different quadrature rules, being the quadrature points part of the input data of our NN, along with the physical points of the domain. For computing the derivatives, we use automatic differentiation, except in some specific cases, where we employ finite differences. These cases are explicitly indicated throughout the text.

Fig. 1 shows a schematic graph of the described NN architecture. Our software is developed in Python and we use the library *Tensorflow 2.0*.

To train the NN, we replace in Eqs. (7) or (10) the search space V by the learnable parameters θ included in our NN. The result of the minimization is a function $u_{NN}(x, \tilde{\theta})$, where $\tilde{\theta}$ are the optimal learnable parameters encountered as a result of the training. For simplicity, in the following we abuse notation and use the symbol u_{NN} to denote also the solution $u_{NN}(x, \tilde{\theta})$ of our minimization problem.

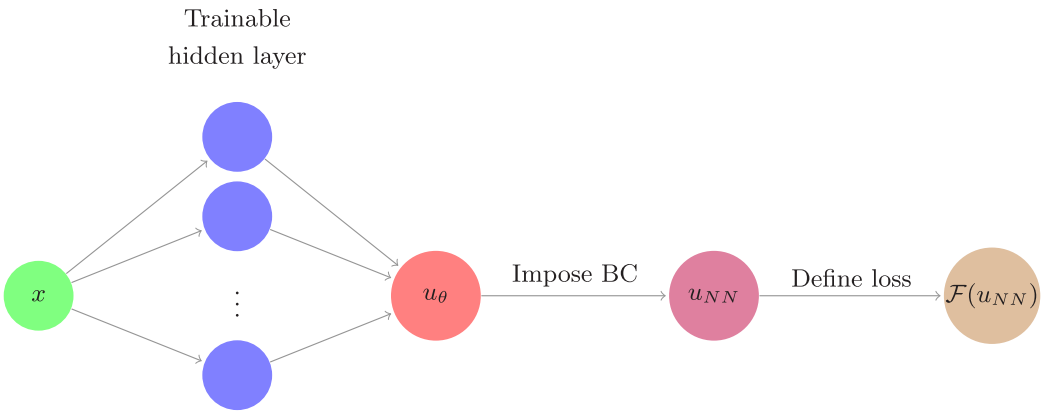


Fig. 1. Our implementation scheme.

5. Quadrature rules

We approximate our integrals from Eqs. (6) and (9) using a quadrature rule of the form

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n} \omega_i f(x_i), \tag{12}$$

where ω_i are the weights and x_i are the quadrature points. Examples of quadrature rules that follow the above formula include trapezoidal rule and Gaussian quadrature rules [42]. We classify these quadrature rules into two groups: (1) those that only employ points from the interior of the interval; and (2) those that evaluate the solution at one extreme point or more (a or b). Integration rules within the later group (e.g., the trapezoidal rule) are inadequate for our minimization problems because the integrand can be infinite at the boundary points in the case of singular solutions. Thus, we focus on quadrature rules that only evaluate the solution at interior points of the domain, with a special focus on Gaussian quadrature rules.

5.1. Illustration of quadrature problems in neural networks

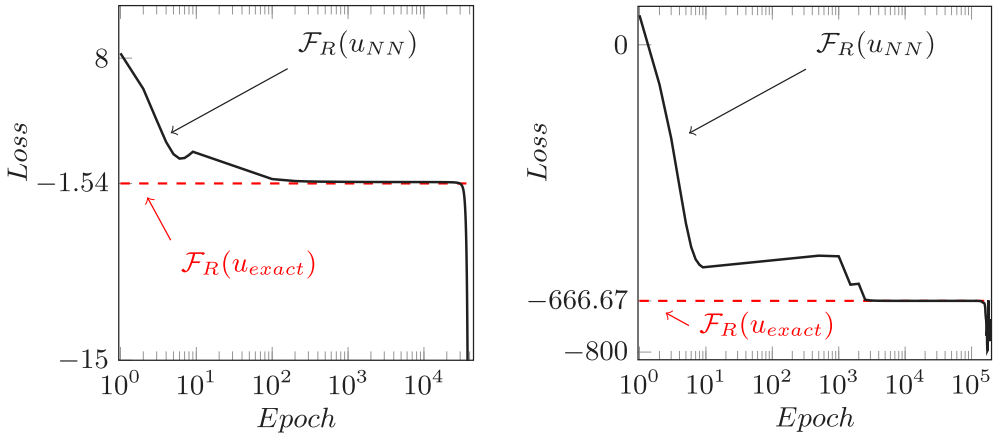
5.1.1. Ritz method

We consider the two model problems from Section 2. We approximate $u(x)$ using the Ritz method. Thus, we search for a NN that minimizes the loss functional given by Eq. (6). Our NN has one hidden layer with 10 neurons (31 trainable weights). We use automatic differentiation to compute the derivatives and a three-point Gaussian quadrature rule to approximate the integrals within each element. We select the Stochastic Gradient Descent (SGD) optimizer. For model problem 1, we discretize our domain with four equal-size elements and execute 40,000 iterations during the optimization process. For model problem 2, we discretize our domain with ten equal-size elements and execute 200,000 iterations during the optimization process.

Figs. 2(a) and 2(b) describe the loss evolution of the training process. We obtain a lower loss than the optimum one, which corresponds to the loss computed analytically using the exact solution (i.e., $\mathcal{F}_R(u_{exact})$). Therefore, we are obtaining a solution that delivers a "better" loss than the exact solution, and so we know that our solution must fail to reasonably approximate the exact one.

Figs. 3 and 4 compare the approximate and exact solutions. We observe a disastrous NN approximation due to quadrature errors. Figs. 3(b) and 4(b) show that the gradient is (almost) zero at the training (quadrature) points of the first interval. Therefore, this value minimizes the numerical approximation of (u', u'). This behavior allows the approximated solution to reach larger values in the first interval, and consequently maximizing the term (f, u), and minimizing the total loss:

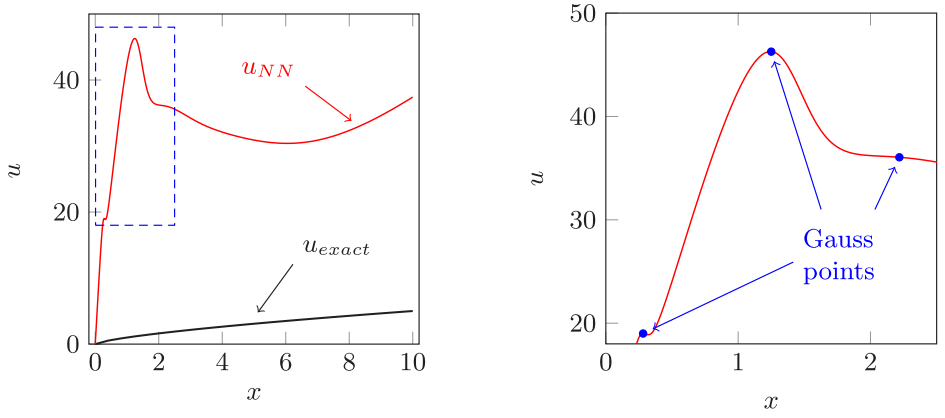
$$\mathcal{F}_R(u_{NN}) = \frac{1}{2} \underbrace{(u'_{NN}, u'_{NN})}_{\sum_{q_i} \omega_i (u'_{NN})^2 \approx 0} - \underbrace{(f, u_{NN})}_{\approx \infty} - (g, u_{NN})_{\Gamma_N} \simeq -\infty$$



(a) Model problem 1.

(b) Model problem 2.

Fig. 2. Loss evolution of the training process for our two model problems.



(a) Exact and approximate solutions.

(b) Approximate solution in the interval $[0, 2.5]$. The Gauss quadrature points corresponding to the first element are indicated in blue.

Fig. 3. Exact vs approximate Ritz method solutions of model problem 1 using four elements for evaluation of $\mathcal{F}_R(v)$ and a NN with 31 weights.

The described quadrature errors can be interpreted as overfitting over a condition of the problem given by the energy of the solution.

5.1.2. Least squares method

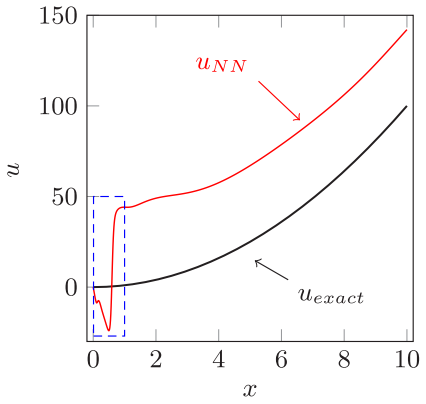
We now consider the following one-dimensional problem:

$$\begin{cases} -u''(x) = 0 & x \in (0, 1), \\ u(0) = u'(1) = 0, \end{cases} \tag{13}$$

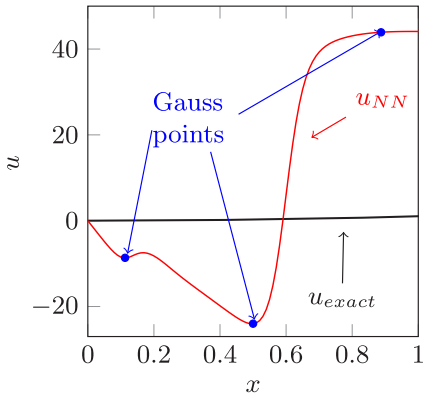
where the exact solution is $u(x) = 0$. We can easily construct an approximating function u_{NN} that satisfies Eq. (13) at the three considered Gaussian points and minimizes Eq. (9), while still being a poor approximation of the exact solution due to quadrature errors. Fig. 5 shows an example.

6. Integral approximation

We now describe four different methods to improve the integral approximations.



(a) Exact and approximate solutions.



(b) Approximate solution in the interval $[0, 1]$. The Gauss quadrature points corresponding to the first element are indicated in blue.

Fig. 4. Exact vs approximate Ritz method solutions of model problem 2 using ten elements for evaluation of $\mathcal{F}_R(v)$ and a NN with 31 weights.

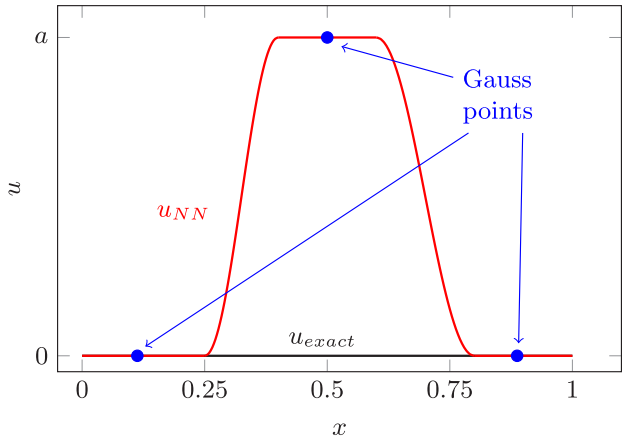


Fig. 5. Exact ($u_{exact} = 0$) and approximated solution of problem given by Eq. (13) and solved with the LS method.

6.1. Monte Carlo integration

We consider the following Monte Carlo integral approximation over a set of points $x_i \in (a, b)$,

$$\int_a^b f(x)dx \approx \frac{(b-a)}{n} \sum_{i=1}^{n} f(x_i), \qquad x_i \in (a, b) \;\; \forall i = \{1, \ldots, n\} \tag{14}$$

In the above, points x_i are randomly selected [43]. While this method is useful for higher-dimensional integrals, for lower dimensions (1D, 2D, 3D) the computational cost is high since the value of the integral approximation converges as $1/\sqrt{n}$ [44].

6.2. Piecewise-polynomial approximation

We replace the original NN u_{NN} by a piecewise-polynomial approximation $u^*_{NN,\cdot}$, where $\cdot$ represents the number of pieces of our piecewise-linear interpolator. This approximation can be exactly differentiated (e.g., via finite

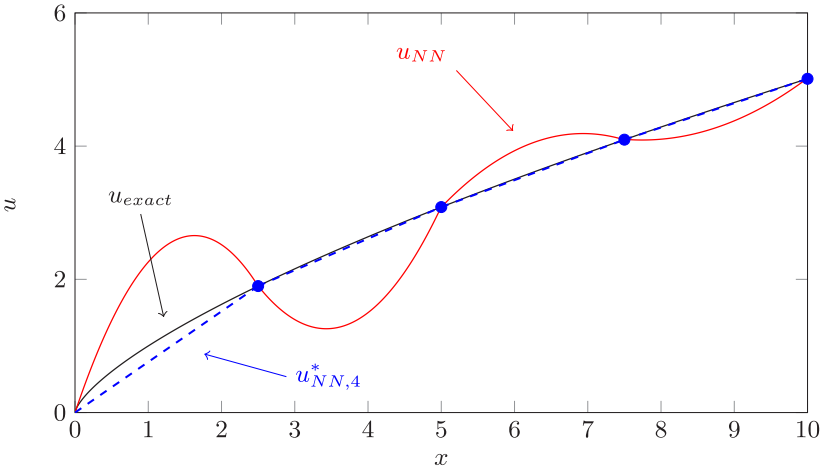


Fig. 6. Neural Network approximation u_{NN} and its four piecewise-linear element approximation $u^*_{NN,4}$.

differences) and integrated (via a Gaussian quadrature rule). Fig. 6 shows an example of the aforementioned case when we train a NN using four elements with linear approximations within each element.

This method controls quadrature errors. However, it is inadequate for high-dimensional problems as we need a mesh that is difficult to implement and integration becomes time consuming.

6.3. Adaptive integration

We first consider a training dataset over the interval (a, b) by taking an equidistant partition of n elements. Then, we define the validation set as a global h-refinement of the training dataset. Fig. 7 shows an example of a training and the corresponding validation datasets. Then, for each element of the training mesh (e.g., E_1 in Fig. 7), we compare the numerical integral over that element vs the sum of the integrals over the two corresponding elements on the validation dataset (in our case, $E_1^1 + E_1^2$). If the integral values differ by more than a stipulated tolerance, we h-refine the training element, and we upgrade the validation dataset so it is built as a global h-refinement of the training dataset. This process is described in Algorithm 1. Fig. 7 also shows a training and the corresponding validation dataset after refining the first and third elements.

We are able to control the quadrature errors by adding new quadrature points to the training dataset. However, the simplest way to implement such method is by using meshes, which posses a limitation on high-dimensional integrals. As an alternative to generating a mesh, one can randomly add points to the training set. This entails difficulties when designing an adaptive algorithm.

In the same way that we propose an h-adaptive method, we can also work with p-adaptivity [45] or a combination of them (e.g., hp-adaptivity [46]).

6.4. Regularization methods

We now introduce a problem-specific regularizer designed to control the quadrature error.

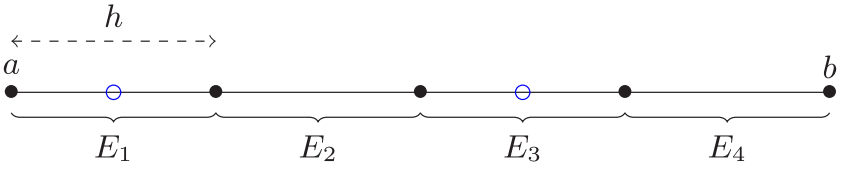
In a one-dimensional setting, we consider the integral functional $\mathcal{F}_R$ as given in (6), and its approximation via a midpoint rule, $\hat{\mathcal{F}}_R$, given by

$$\hat{\mathcal{F}}_R(u) = \frac{b-a}{N} \sum_{j=1}^{N} \left(\frac{1}{2} |u'(x_j)|^2 - f(x_j)u(x_j) \right) + g(b)u(b) + g(a)u(a). \tag{15}$$

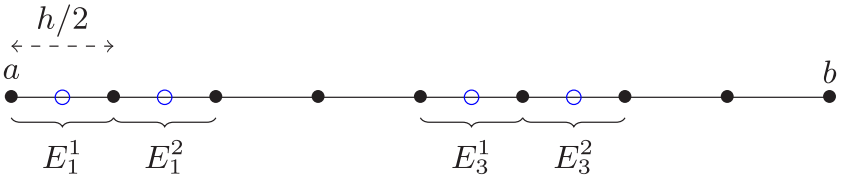
We note that $g = 0$, except where the Neumann condition is imposed. While we focus on the Ritz method, a similar heuristic can be applied to the Least Squares method.

Algorithm 1: Adaptive integration method

Generate a training dataset;
Generate the corresponding validation dataset;
Set tolerance ϵ and maximum iteration number i_{max};
while $i < i_{max}$ **do**
  for $j = 1, \cdots, n$ **do**
    Compute integral values I_j over the training dataset of elements E_j;
    Compute integral values I_j^1,I_j^2 over the validation dataset of elements E_j^1,E_j^2 ;
    if $|(I_j^1 + I_j^2) - I_j| > \epsilon$ **then**
      h-refine the E_j-th element of the training set;
      h-refine the E_j^1-th and E_j^2-th elements of the validation set;
    else
      continue;
    end
  end
  $i = i + 1$;
end



(a) Training set



(b) Validation set

Fig. 7. Black points (dots) correspond to the original (a) training/(b) validation partitions and blue points (circles) are the points added by the refinement performed in the first and third elements.

We introduce a function $\mathcal{R}$ that depends on the learnable parameters θ of a given Neural Network u_{NN}, such that for any Neural Network with a given architecture,

$$|\mathcal{F}_R(u_{NN}) - \hat{\mathcal{F}}_R(u_{NN})| < \mathcal{R}(\theta). \tag{16}$$

If we then consider a loss function $\mathcal{L}$ given by

$$\mathcal{L}(\theta) = \hat{\mathcal{F}}_R(u_{NN}) + \mathcal{R}(\theta), \tag{17}$$

then we may be able to improve the approximation of the quadrature rule, as the loss contains a term that by design controls the quadrature error.

For simplicity, we consider only the case of a single-layer network with a one-dimensional input, and the midpoint rule for calculating the integral over a uniform partition of (a, b). We consider a mid-point rule as in (15), and define the interval length $\delta = \frac{b-a}{N}$ and intervals $I_j = \left(\frac{\delta}{2} + x_j, x_j + \frac{\delta}{2}\right)$. We estimate the error of the midpoint rule

to integrate F as

$$\begin{aligned}\left|\int_a^b F(x)\,dx - \sum_{j=1}^{N} F(x_j)\delta\right| &= \left|\sum_{i=1}^{N}\int_{x_j-\frac{\delta}{2}}^{x_j+\frac{\delta}{2}} (F(x)-F(x_j))\,dx\right| \\ &\leq \sum_{i=1}^{N}\int_{x_j-\frac{\delta}{2}}^{x_j+\frac{\delta}{2}} \left|F(x)-F(x_j)\right|\,dx \\ &\leq \sum_{i=1}^{N}\max_{t\in I_j}|F'(t)|\int_{x_j-\frac{\delta}{2}}^{x_j+\frac{\delta}{2}}|x-x_j|\,dx \\ &= \frac{\delta^2}{4}\sum_{i=1}^{N}\max_{t\in I_j}|F'(t)|.\end{aligned} \tag{18}$$

This estimate scales as $\mathcal{O}(\frac{1}{N})$ for fixed F, and thus, for a large number of integration points, we expect the estimate to be sufficiently accurate and to avoid "overdamping" of the loss.

With (18) in mind, we estimate the local Lipschitz constants of the integrand as in (6). The numerical estimation of the Lipschitz constants of NNs has attracted attention, as they form a way of estimating the generalizability of a Neural Network, and have been used in the training process as a way to encourage accurate generalization [47–49]. As we are dealing with loss functions that involve *derivatives* of the Neural Network, we however need estimates of higher order derivatives of u_{NN}. The approach that we employ is similar in spirit to the work of [26] for obtaining *a posteriori* error estimates in PINNs.

Despite the arithmetic complications involved in calculating $\mathcal{R}$, conceptually the idea reduces to an application of Taylor's theorem. On a single interval of integration I_j, we have that for every x, there exists some ξ_x so that

$$|F'(x)| = |F'(x_i) + (x-x_i)F''(\xi_x)| \leq |F'(x_i)| + \frac{\delta}{2}\|F''\|_\infty. \tag{19}$$

We then find $\mathcal{R}$ using a combination of local and global estimates for the derivatives of the integrand corresponding to simple pointwise evaluations at the integration points and global estimates involving the Neural Network weights. The exact calculation of the estimates that produce the regularizer $\mathcal{R}$ are tedious and deferred to the Appendix, with $\mathcal{R}$ given in (A.16) via expressions in (A.5) and (A.9).

7. Numerical results

In this section, we test some of the alternatives proposed in Section 6 to overcome the quadrature problems. Specifically, we solve the two model problems from Section 2 using: (a) a piecewise-linear approximation of the NN, and (b) an adaptive integration method. We also solve model problem 2 using regularization methods.

For the cases of piecewise-linear approximation and adaptive integration, we use a NN architecture composed of one hidden layer with ten neurons and a *sigmoid* activation function. Moreover, we select SGD as the optimizer. In the case of regularization, we use a *hyperbolic tangent* activation function with the Adam optimizer.

7.1. Piecewise-linear approximation

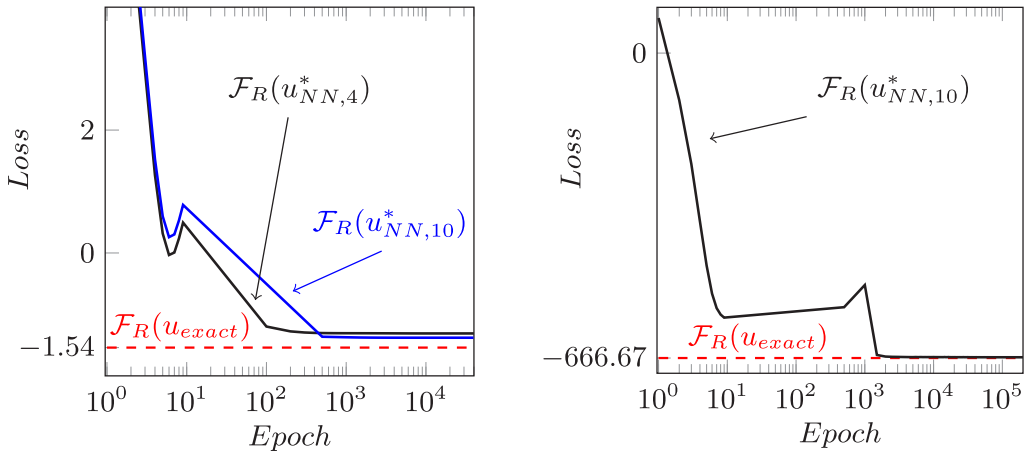
We select a piecewise-linear approximation of the NN as our approximate solution. We compute the gradients using Finite Differences and the integrals using a one-point Gaussian quadrature rule (i.e., the midpoint rule). For model problem 1, we use two different uniform partitions composed of four and ten elements, and execute 40,000 iterations. For model problem 2, we use a uniform partition composed of ten elements and execute 200,000 iterations.

Figs. 8(a) and 8(b) show that the loss converges to the loss of the exact solution. We also observe better results as we increase the number of elements, as physically expected. Figs. 9(a) and 9(b) show the corresponding solutions, which are consistent with the loss evolutions displayed in Figs. 8(a) and 8(b).

Table 1 shows the loss value of the exact solution, of the optimum piecewise/linear solution, and of the obtained DNN piecewise-linear solution. Since we are solving a 1D Laplace problem, the best piecewise-linear approximation

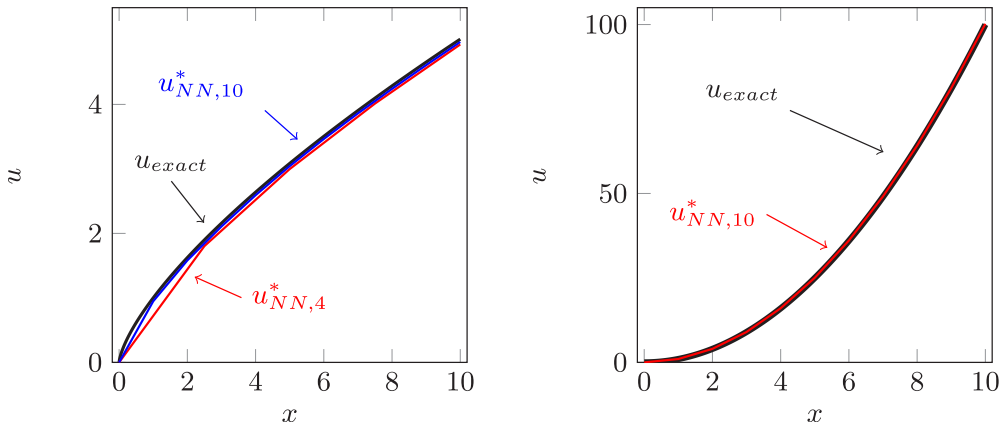
Table 1
Loss values of the exact solution $\mathcal{F}_R(u_{exact})$, optimum piecewise-linear solution $\mathcal{F}_R(\tilde{u}^*_{NN,\cdot})$ (for a four and a ten equidistant element partition), and obtained piecewise-linear solution $\mathcal{F}_R(u^*_{NN,\cdot})$ using a DNN.

	$\mathcal{F}_R(u_{exact})$	$\mathcal{F}_R(\tilde{u}^*_{NN,4})$	$\mathcal{F}_R(u^*_{NN,4})$	$\mathcal{F}_R(\tilde{u}^*_{NN,10})$	$\mathcal{F}_R(u^*_{NN,10})$
Model problem 1	-1.54	-1.36	-1.31	-1.41	-1.38
Model problem 2	-666.67	–	–	-665	-664.99



(a) Model problem 1. (b) Model problem 2.

Fig. 8. Loss evolution of the training process for our two model problems when we use a piecewise-linear approximation of the NN.

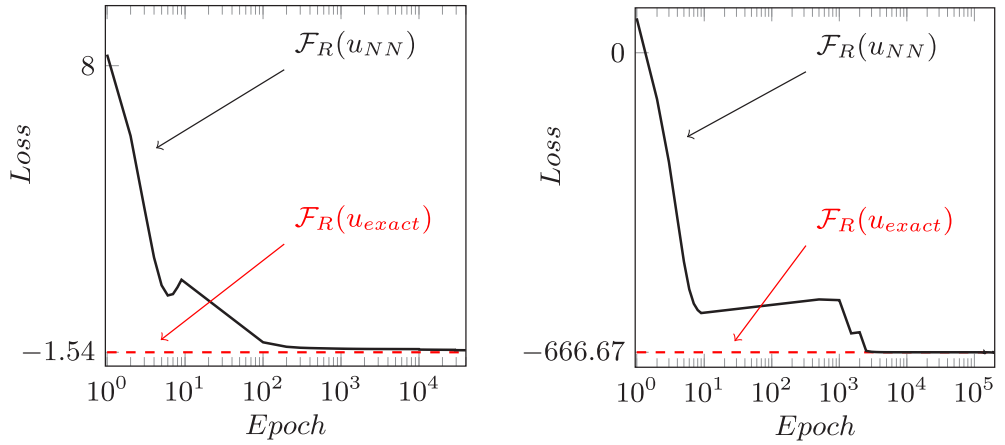


(a) Model problem 1. (b) Model problem 2.

Fig. 9. Ritz method solution when we use a piecewise-linear approximation of the NN to solve the problem.

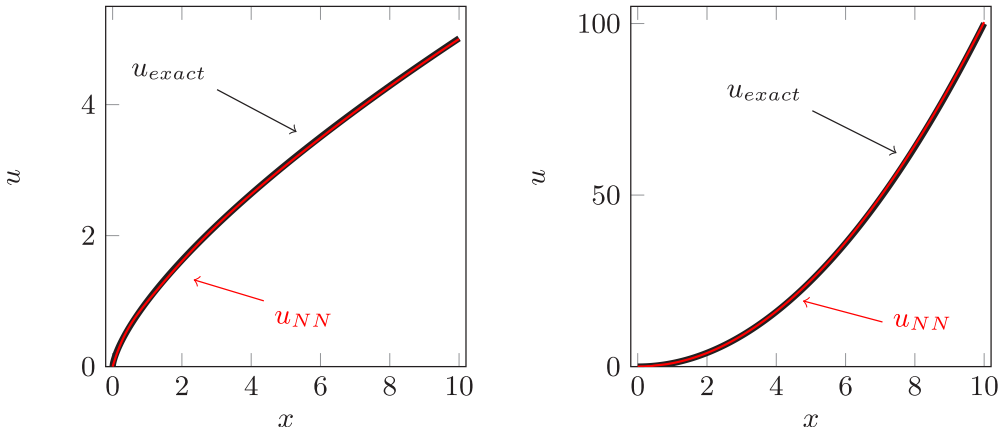
of the solution is its interpolator at the vertex nodes. For model problem 1, we observe that we do not obtain the best piecewise-linear solution. For model problem 2, we reach the optimum piecewise-solution.

While the use of a piecewise-linear approximation overcomes the quadrature problems, the convergence is limited to $\mathcal{O}(h)$, where h is the element size [50]. To increase this speed, it is possible to consider different piecewise-polynomial approximations, including the use of r-adaptive algorithms [51].



(a) Model problem 1. (b) Model problem 2.

Fig. 10. Loss evolution of the training process for our two model problems when using adaptive integration.



(a) Model problem 1. (b) Model problem 2.

Fig. 11. Ritz method solution when using adaptive integration.

7.2. Adaptive integration

We compute the gradients using automatic differentiation and the integrals using a three-point Gaussian quadrature rule. As in previous examples, we start the training with a uniform partition of four elements for model problem 1 and of ten elements for model problem 2. We select a half-size partition of the training dataset for validation. For model problem 1, we compare the integral values of the training and validation sets (i.e., we execute Algorithm 1) every 1000 iterations, with an error tolerance of 0,01; for model problem 2, we compare every 10,000 iterations, with an error tolerance of 10. The tolerance is selected as a small percentage of the value of the loss. In both cases, we do not execute the adaptive algorithm in the first 10,000 iterations.

Figs. 10(a) and 10(b) show that the loss converges to the optimum value. As explained in Section 6, the adaptive integration algorithm refines the training dataset. For model problem 1, the algorithm performs two refinements in the first interval. For model problem 2, one refinement occurs in the first interval. The adaptive integration algorithm automatically selects the first interval for refinement, where overfitting was taking place in Figs. 3 and 4. Figs. 11(a) and 11(b) show the corresponding solutions. We observe that the approximate solutions properly approximate the exact ones.

The extrapolation of this method to higher dimensions (2D or 3D) requires higher-dimensional discretizations and quadrature rules.

7.3. Regularization methods

We do not apply the regularization method to model problem 1 as the method requires sufficient regularity in order to provide the necessary estimates in the calculation of $\mathcal{R}$. Since the solution is singular at $x = 0$, the necessary Lipschitz bounds on the integral functional cannot be obtained within this framework. Instead, we aim to demonstrate that for problems that are sufficiently regular, our technique can avoid overfitting, and leave open the question as to how one may adapt the technique to singular problems for future work. We thus consider model problem 2. We propose the loss defined via

$$\mathcal{L}(\theta) = \hat{\mathcal{F}}_R(u_{NN}) + \mathcal{R}(\theta). \tag{20}$$

Explicitly,

$$\hat{\mathcal{F}}_R(u_{NN}) = \frac{10}{N}\sum_{j=1}^{N} \frac{1}{2}|u'_{NN}(x_j)|^2 - 2u_{NN}(x_j) - 20u_{NN}(20), \tag{21}$$

where $x_j = \frac{10}{N}\left(i - \frac{1}{2}\right)$.

7.3.1. Experiment 1

We consider $N = 50$ points, and a single layer network with $M = 10$ neurons. We use the Adam optimizer with learning rate 10^{-2}. We solve model problem 2 with two losses: with and without regularization. In both cases, we measure the metrics $\mathcal{L}$, $\mathcal{R}$, and $\hat{\mathcal{F}}_R$. For validation, we use an equidistant partition of (0, 10) with 49 points, so that we still use a midpoint rule but with different integration points.

Fig. 12 shows the results without regularization. As expected, we see in Fig. 12(a) that the approximation is poor due to overfitting, which is most notable around $x = 0$ and attained within 5000 epochs. Via the provided plots we can observe the beginning of overfitting in two distinct manners. First, we observe in Fig. 12(c) that the value of $\hat{\mathcal{F}}_R$ evaluated over the validation data begins to diverge from the value on the training data, becoming apparent at around 1000 epochs. We also see this behavior reflected in the evolution of $\mathcal{R}$ in Fig. 12(d), with its most dramatic increase beginning around the same iteration. This rapid increase also provokes an increase in $\mathcal{L}$, as seen in Fig. 12(b). This also indicates that even if $\mathcal{R}$ is not used as part of the training process, its increase could be used as a metric to identify overfitting.

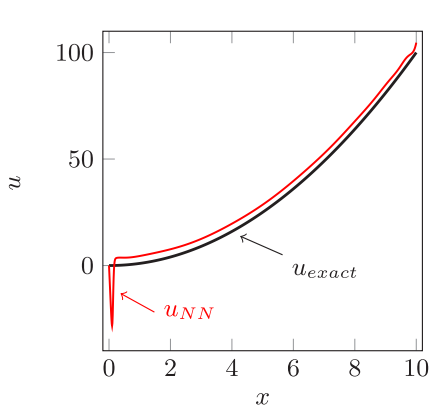
Fig. 13 describes the results with regularization and we observe a different behavior. The approximation is generally good, and we do not see any signs of overfitting within 10^5 epochs, as shown in Fig. 13(a). In particular, the values of $\hat{\mathcal{F}}_R$ at the training and validation data remain consistent in Fig. 13(c). Throughout Fig. 13 we see that within 10^5 epochs all metrics appear to have converged to a limiting value. We obtain final values $\mathcal{L} \approx -644.22$, $\hat{\mathcal{F}}_R \approx -666.07$, $\mathcal{R} \approx 24.8$. We recall that the true energy of the exact solution is $\mathcal{F}_R(u_{exact}) \approx -666.667$, which suggests the quadrature rule is accurate. Notice that in the case without regularization, before overfitting became apparent, $\mathcal{R}$ had already attained values of around 1000, which is far larger than the value of $\mathcal{R}$ at the obtained solution when regularization was used.

7.3.2. Experiment 2

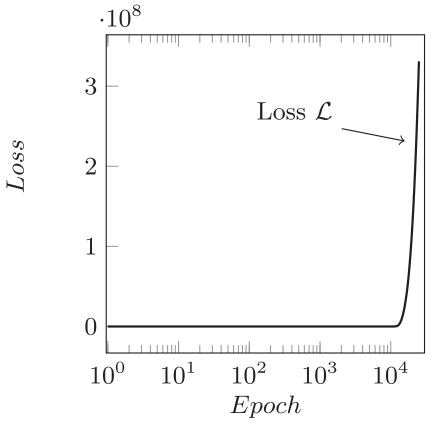
We now consider a smaller N. As we expect $\mathcal{R}$ to scale as $\frac{1}{N}$, we anticipate a more adverse effect when N is small. To view this, we consider the same problem of Experiment 1, where we now select $N = 20$ integration points. We consider $M = 10$ neurons and minimize our problem using the Adam optimizer with a learning rate of 10^{-2}. As before, we consider the cases with and without regularization.

Fig. 14 presents the loss evolution without regularization. We observe overfitting, which is accompanied by divergence of the loss on the validation dataset, as well as a rapid increase in $\mathcal{R}$, with these features visible within 5000 epochs.

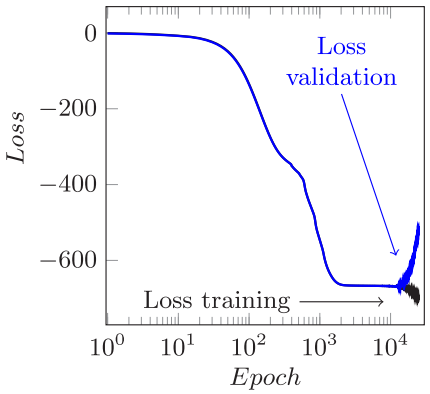
Fig. 15 presents the results with regularization. We observe no signs of overfitting, with the validation and training loss remaining close in Fig. 15(b). All metrics appear to have converged to a limiting value within 10^4 epochs.



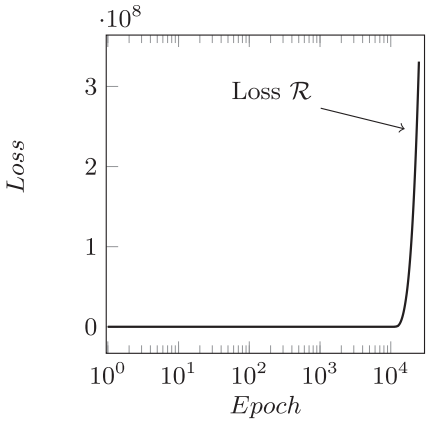
(a) Exact and approximated solutions.



(b) Evolution of $\mathcal{L}$ during training.



(c) Evolution of $\hat{\mathcal{F}}_R$ during training.



(d) Evolution of $\mathcal{R}$ during training.

Fig. 12. The solution and training information for Experiment 1 without regularization.

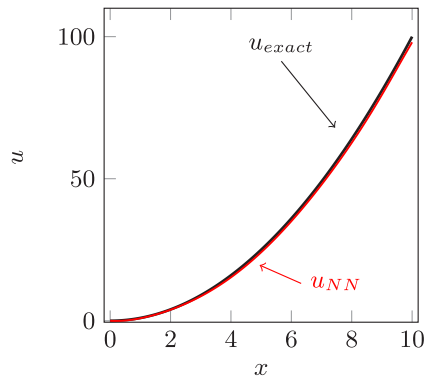
However, the large value of $\mathcal{R}$ at the found solution (approximately 140) has substantially changed the optimization problem so that the obtained minimizer is far from the desired solution. The final value of $\hat{\mathcal{F}}_R$ is around -622, which is far from the desired value of -666.67. This experiment highlights the fact that the regularizer becomes more effective when a large number of integration points are used.

8. Conclusions & future work

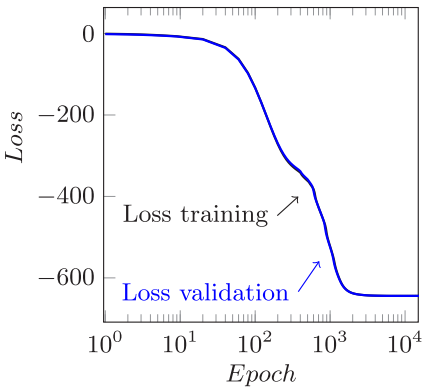
We first illustrated how quadrature errors can destroy the quality of the approximated solution when solving PDEs using DL methods. Thus, it is crucial to select an adequate method to overcome the quadrature problems. Herein, we proposed four different alternatives: (a) Monte Carlo integration, (b) piecewise-polynomial approximations of the output of the Neural Network, (c) adaptive integration, and (d) regularization methods. We discussed the advantages and limitations of each of these methods, and we illustrated their performance via simple 1D numerical examples.

In high dimensions, Monte Carlo integration methods are the best choice. Regularizer methods are another option, but they are problem dependent and they need to be derived for each different architecture. Moreover, they require further analysis for highly nonlinear integrands. Furthermore, they are limited only to sufficiently smooth integral functionals. In addition, more complex NN architectures (which should be needed in higher dimension) will hinder the derivation of $\mathcal{R}$.

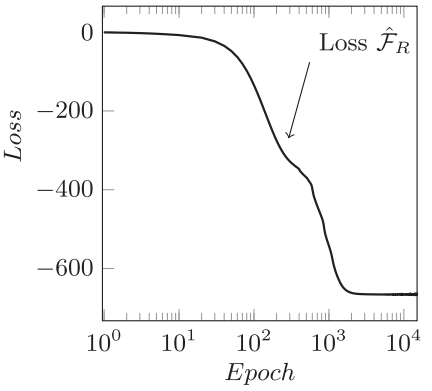
In low dimensions (three or below), Monte Carlo integration is not competitive because of its low convergence speed. In these cases, adaptive integration exhibits faster convergence. In the cases of piecewise-linear approximation



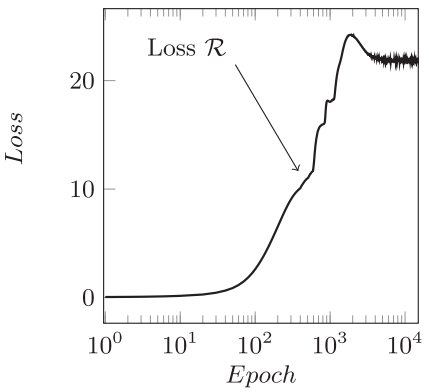
(a) Exact and approximated solutions.



(b) Evolution of $\mathcal{L}$ during training.



(c) Evolution of $\hat{\mathcal{F}}_R$ during training.



(d) Evolution of $\mathcal{R}$ during training.

Fig. 13. The solution and training information for Experiment 1 with regularization.

and regularizers, we are also able to overcome the quadrature problems, but the convergence speed is often slower than with adaptive integration.

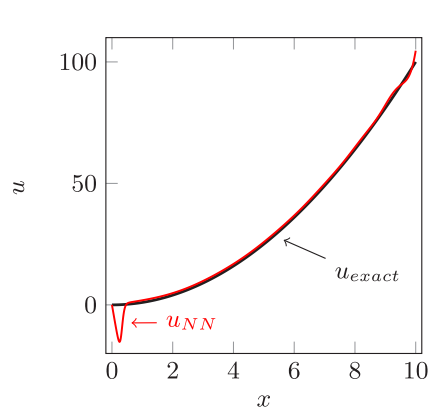
Possible future research lines of this work are: (a) to implement adaptive integration method in 2D and 3D, (b) to apply *r-adaptivity* methods with piecewise-polynomial approximations, and (c) to implement regularizers for high dimensional problems involving more complex NN architectures.

Declaration of competing interest

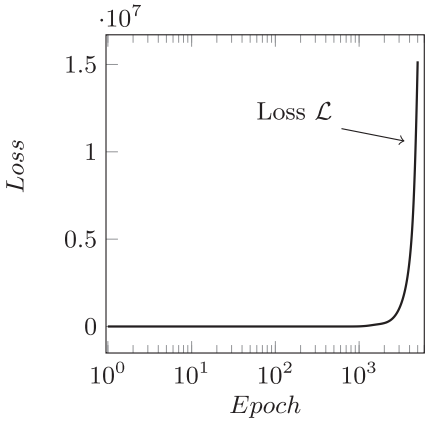
The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

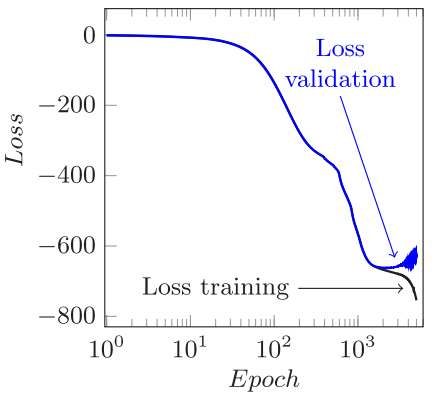
This work has received funding from: the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No. 777778 (MATHROCKS); the European Regional Development Fund (ERDF) through the Interreg V-A Spain-France-Andorra program POCTEFA 2014–2020 Project PIXIL (EFA362/19); the Spanish Ministry of Science and Innovation projects with references PID2019-108111RB-I00 (FEDER/AEI), PDC 2021-121093-I00, and PID2020-114189RB-I00 and the "BCAM Severo Ochoa" accredidation of excellence (SEV-2017-0718); and the Basque Government, Spain through the three Elkartek projects 3KIA (KK-2020/00049), EXPERTIA (KK-2021/000 48), and SIGZE (KK-2021/00095), the Consolidated Research Group MATHMODE (IT1294-19) given by the Department of Education, and the BERC 2022–2025 program.



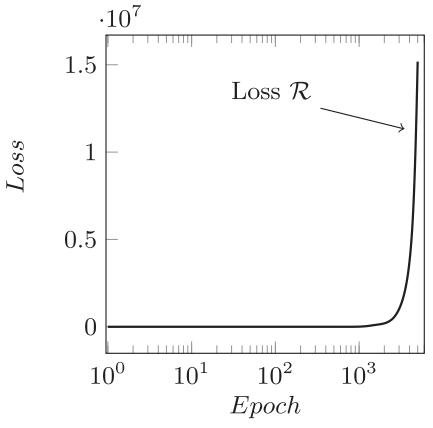
(a) Exact and approximated solutions.



(b) Evolution of $\mathcal{L}$ during training.



(c) Evolution of $\hat{\mathcal{F}}_R$ during training.



(d) Evolution of $\mathcal{R}$ during training.

Fig. 14. The solution and training information for Experiment 2 without regularization.

Appendix. Estimation of the regularizer

Following the heuristic of Section 6.4, we derive an expression for $\mathcal{R}$ that may be used as a regularizer. The necessary steps are:

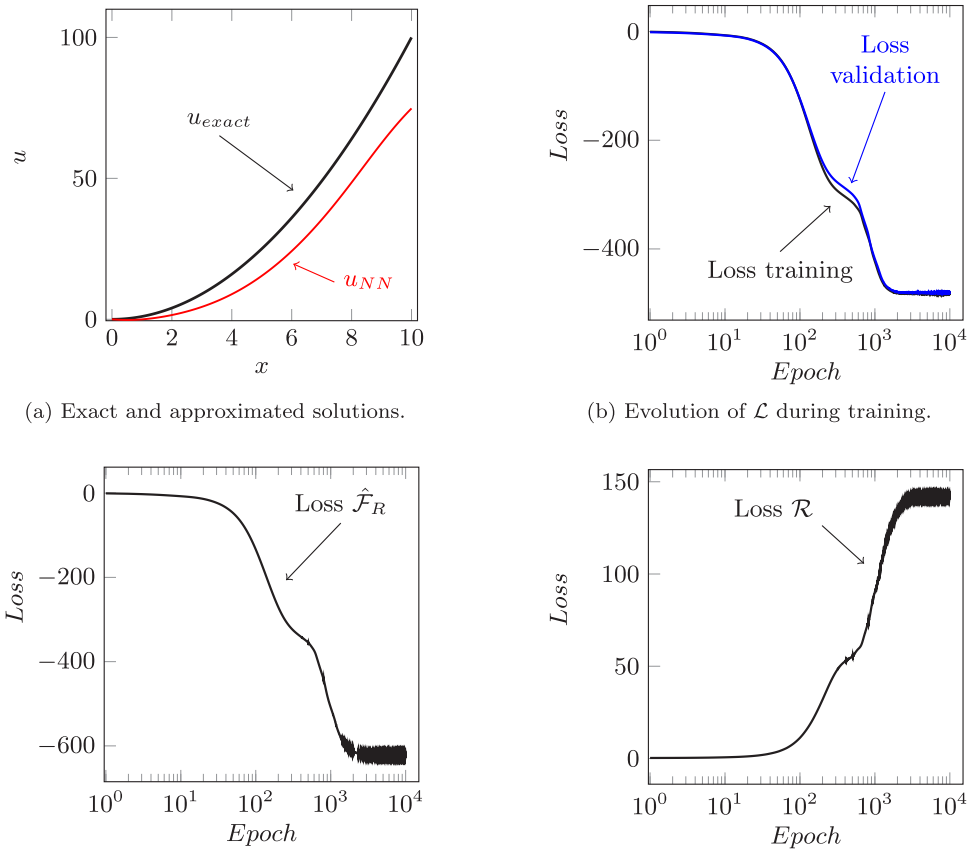
1. Using the chain rule, we find global upper bounds for the derivatives of a simple Neural Network in terms of the weights.
2. Via Taylor's theorem with remainder and using the global estimates for simple networks, we find local estimates of the derivatives of NNs with a cutoff function to ensure a homogeneous Dirichlet condition.
3. Using local estimates for the derivatives of a NN, we find local Lipschitz estimates for integrands corresponding to the Ritz method.

We tackle each of these estimations in the following subsections.

A.1. Global estimates for derivatives of a single layer network

Let $\hat{u}_{NN}$ be a single layer Neural Network. We write it in the form

$$\hat{u}_{NN}(x) = b^1 + \sum_{i=1}^{M} A_i^1 \sigma(A_i^0 x + b_i^0). \tag{A.1}$$



(a) Exact and approximated solutions.

(b) Evolution of $\mathcal{L}$ during training.

(c) Evolution of $\hat{\mathcal{F}}_R$ during training.

(d) Evolution of $\mathcal{R}$ during training.

Fig. 15. The solution and training information for Experiment 2 with regularization.

for weights $\theta = (b^1, A^1, b^0, A^0)$ and an activation function σ. We assume that σ has globally bounded derivatives, so that $\|\sigma^{(n)}\|_\infty$ is finite for every $n = 0, 1, 2....$

A simple application of the triangle quality and that σ is bounded gives that

$$\begin{aligned} |\hat{u}_{NN}(x)| &\leq |b^1| + \sum_{i=1}^{M} |A_i^1 \sigma(A_i^0 x + b_i^0)| \\ &\leq |b^1| + \sum_{i=1}^{M} |A_i^1| \|\sigma\|_\infty \end{aligned} \tag{A.2}$$

The derivatives of $\hat{u}_{NN}$ are

$$\hat{u}_{NN}^{(n)}(x) = \sum_{i=1}^{M} A_i^1 (A_i^0)^n \sigma^{(n)}(A_i^0 x + b_i^0), \tag{A.3}$$

for $n \geq 1$. Similarly, this gives the immediate estimate that

$$\left|\hat{u}_{NN}^{(n)}(x)\right| \leq \sum_{i=1}^{M} |A_i^1| |A_i^0|^n \|\sigma^{(n)}\|_\infty. \tag{A.4}$$

Thus, we define the global upper bounding function $\mathcal{R}^1(\theta; n)$ by

$$\mathcal{R}^1(\theta; n) = \begin{cases} |b^1| + \|\sigma\|_\infty \sum_{i=1}^M |A_i^1| & n = 0, \\ \|\sigma^{(n)}\|_\infty \sum_{i=1}^M |A_i^1||A_i^0|^n & n \geq 1, \end{cases} \tag{A.5}$$

which gives the global upper bound

$$\left|\hat{u}_{NN}^{(n)}(x)\right| \leq \mathcal{R}^1(\theta; n) \tag{A.6}$$

for every x.

A.2. Local derivative estimation of a single layer neural network with cutoff function

Next, we consider the commonly considered case where our Neural Network admits a cutoff function to ensure a Dirichlet boundary condition, and turn to the case of *local* estimation. Explicitly, we take $u_{NN}(x) = \hat{u}_{NN}(x)\phi(x)$, where ϕ is zero at the homogeneous Dirichlet condition, and $\hat{u}_{NN}$ is as in (A.1). We presume that ϕ has bounded derivatives; i.e. $\|\phi^{(k)}\|_\infty$ is finite for $k = 0, 1, \ldots$. Let x be in the interval $I_j = \left(x_j - \frac{\delta}{2}, x_j + \frac{\delta}{2}\right)$. Then, we have

$$\begin{aligned} u_{NN}^{(n)}(x) &= u_{NN}^{(n)}(x_j) + (x - x_j)u_{NN}^{(n+1)}(\xi_x) \\ &= u_{NN}^{(n)}(x_j) + (x - x_j)\sum_{k=0}^{n+1} \binom{n+1}{k} \hat{u}_{NN}^{(k)}(\xi_x)\phi^{(n+1-k)}(\xi_x) \end{aligned} \tag{A.7}$$

via Taylor's theorem with remainder and the product rule for higher order derivatives. Employing the global upper bounds $\mathcal{R}^1(\theta; n)$ from (A.5), we have that for $x \in I_j$,

$$\begin{aligned} \left|u_{NN}^{(n)}(x)\right| &\leq \left|u_{NN}^{(n)}(x_j)\right| + \frac{\delta}{2}\sum_{k=0}^{n+1} \binom{n+1}{k} |\hat{u}_{NN}^{(k)}(\xi_x)||\phi^{(n+1-k)}(\xi_x)| \\ &\leq \left|u_{NN}^{(n)}(x_j)\right| + \frac{\delta}{2}\sum_{k=0}^{n+1} \binom{n+1}{k} \mathcal{R}^1(\theta; k)\|\phi^{(n+1-k)}\|_\infty. \end{aligned} \tag{A.8}$$

Thus, we define the second intermediate regularizer as

$$\mathcal{R}^2(\theta; I_j, n) = \left|u_{NN}^{(n)}(x_j)\right| + \frac{\delta}{2}\sum_{k=0}^{n+1} \binom{n+1}{k} \mathcal{R}^1(\theta; k)\|\phi^{(n+1-k)}\|_\infty, \tag{A.9}$$

giving the estimate that for all $x \in I_j$,

$$\left|u_{NN}^{(n)}(x)\right| \leq \mathcal{R}^2(\theta; I_j, n). \tag{A.10}$$

We note that via automatic differentiation, $u^{(n)}$ may be evaluated at the training data x_j.

A.3. Application to integral errors

Our aim is to estimate the error in the integral functional

$$\mathcal{F}(u_{NN}) = \int_a^b \frac{1}{2}|u'_{NN}(x)|^2 - f(x)u_{NN}(x)\,dx - g(a)u_{NN}(a) - g(b)u_{NN}(b), \tag{A.11}$$

when approximated by a simple quadrature rule

$$\sum_{j=1}^N \left(\frac{1}{2}|u'_{NN}(x_j)|^2 - f(x_j)u_{NN}(x_j)\right)\delta - g(b)u_{NN}(b) - g(a)u_{NN}(a). \tag{A.12}$$

The boundary terms can be calculated in one dimension without quadrature error and thus we ignore their contribution. We estimate the error for the quadrature rule by obtaining Lipschitz bounds of the integrand via

$$\begin{aligned}&\left|\frac{d}{dx}\left(\frac{1}{2}|u'_{NN}(x)|^2 - f(x)u_{NN}(x)\right)\right| \\ &= \left|u'_{NN}(x)u''_{NN}(x) - f'(x)u_{NN}(x) - f(x)u'_{NN}(x)\right| \\ &\leq |u'_{NN}(x)||u''_{NN}(x)| + |f'(x)||u_{NN}(x)| + |f(x)||u'_{NN}(x)|.\end{aligned} \tag{A.13}$$

Estimating the (local) Lipschitz constant of the integrand reduces to estimating (locally) various derivatives of u_{NN}. For $x \in I_j$, we estimate the Lipschitz constant of the integrand via

$$\left|\frac{d}{dx}\left(\frac{1}{2}|u'_{NN}(x)|^2 - f(x)u_{NN}(x)\right)\right| \leq \mathcal{R}^3(\theta; I_j), \tag{A.14}$$

where the regularizer $\mathcal{R}^3(\theta, I_j)$ is given by

$$\mathcal{R}^3(\theta; I_j) = \Big(\mathcal{R}^2(\theta; I_j, 1)\mathcal{R}^2(\theta; I_j, 2) + \|f\|_\infty \mathcal{R}^2(\theta; I_j, 1) + \|f'\|_\infty \mathcal{R}^2(\theta; I_j, 0)\Big). \tag{A.15}$$

We define the final regularizer $\mathcal{R}$ by

$$\mathcal{R}(\theta) = \frac{\delta^2}{4}\sum_{j=1}^{N} \mathcal{R}^3(\theta; I_j), \tag{A.16}$$

which following (18) gives the estimate

$$\begin{aligned}&|\mathcal{F}(u_{NN}) - \hat{\mathcal{F}}(u_{NN})| \\ &= \left|\int_a^b \frac{1}{2}|u'_{NN}(x)|^2 - f(x)u_{NN}(x)\,dx - \sum_{j=1}^{N}\left(\frac{1}{2}|u'_{NN}(x_j)|^2 - f(x_j)u_{NN}(x_j)\right)\delta\right| \\ &\leq \mathcal{R}(\theta).\end{aligned} \tag{A.17}$$

References

[1] S. Ranjan, S. Senthamilarasu, Applied Deep Learning and Computer Vision for Self-Driving Cars: Build autonomous vehicles using deep neural networks and behavior-cloning techniques, Packt Publishing, 2020.

[2] A. Gupta, A. Anpalagan, L. Guan, A.S. Khwaja, Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues, Array 10 (2021) 100057, http://dx.doi.org/10.1016/j.array.2021.100057.

[3] T. Afouras, J.S. Chung, A. Senior, O. Vinyals, A. Zisserman, Deep audio-visual speech recognition, IEEE Trans. Pattern Anal. Mach. Intell. PP (2018) 1, http://dx.doi.org/10.1109/TPAMI.2018.2889052.

[4] M. Alam, M. Samad, L. Vidyaratne, A. Glandon, K. Iftekharuddin, Survey on deep neural networks in speech and vision systems, Neurocomputing 417 (2020) 302–321, http://dx.doi.org/10.1016/j.neucom.2020.07.053.

[5] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, J. Dean, A guide to deep learning in healthcare, Nat. Med. 25 (2019) 24–29, http://dx.doi.org/10.1038/s41591-018-0316-z.

[6] S. Purushotham, C. Meng, Z. Che, Y. Liu, Benchmarking deep learning models on large healthcare datasets, J. Biomed. Inf. 83 (2018) 112–134, http://dx.doi.org/10.1016/j.jbi.2018.04.007.

[7] K. Wu, D. Xiu, Data-driven deep learning of partial differential equations in modal space, J. Comput. Phys. 408 (2020) 109307, http://dx.doi.org/10.1016/j.jcp.2020.109307.

[8] J. Berg, K. Nyström, A unified deep artificial neural network approach to partial differential equations in complex geometries, Neurocomputing 317 (2018) 28–41, http://dx.doi.org/10.1016/j.neucom.2018.06.056.

[9] E. Samaniego, C. Anitescu, S. Goswami, V. 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. Eng. 362 (2020) 112790, http://dx.doi.org/10.1016/j.cma.2019.112790.

[10] L. Ruthotto, E. Haber, Deep neural networks motivated by partial differential equations, J. Math. Imaging Vis. 62 (2020) http://dx.doi.org/10.1007/s10851-019-00903-1.

[11] L. Lu, X. Meng, Z. Mao, G.E. Karniadakis, Deepxde: A deep learning library for solving differential equations, 2020, arXiv:1907.04502.

[12] M. Shahriari, D. Pardo, J.A. Rivera, C. Torres-Verdín, A. Picon, J. Del Ser, S. Ossandón, V. Calo, Error control and loss functions for the deep learning inversion of borehole resistivity measurements, Int. J. Num. Methods Eng. (2020) http://dx.doi.org/10.1002/nme.6593.

[13] I. Brevis, I. Muga, K. Zee, A machine-learning minimal-residual (ML-mres) framework for goal-oriented finite element discretizations, Comput. Math. Appl. 95 (2020) http://dx.doi.org/10.1016/j.camwa.2020.08.012.

[14] M. Paszyński, R. Grzeszczuk, D. Pardo, L. Demkowicz, Deep learning driven self-adaptive hp finite element method, in: Computational Science – ICCS 2021, Springer International Publishing, 2021, pp. 114–121, http://dx.doi.org/10.1007/978-3-030-77961-0_11.

[15] J.R. Hauser (Ed.), Partial differential equations: The finite element method, in: Numerical Methods For Nonlinear Engineering Models, Springer Netherlands, Dordrecht, 2009, pp. 883–987, http://dx.doi.org/10.1007/978-1-4020-9920-5_13.
[16] R. LeVeque, Finite Difference Methods For Ordinary And Partial Differential Equations: Steady-State And Time-Dependent Problems (Classics In Applied Mathematics Classics In Applied Mathemat), Society for Industrial and Applied Mathematics, USA, 2007.
[17] V.P. Nguyen, C. Anitescu, S.P. Bordas, T. Rabczuk, Isogeometric analysis: An overview and computer implementation aspects, Math. Comput. Simul. 117 (2015) 89–116, http://dx.doi.org/10.1016/j.matcom.2015.05.008.
[18] J. Han, A. Jentzen, W. Ee, Solving high-dimensional partial differential equations using deep learning, Proc. Natl. Acad. Sci. 115 (2017) http://dx.doi.org/10.1073/pnas.1718942115.
[19] W. Ee, J. Han, A. Jentzen, Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations, Appear Commun. Math. Stat. 5 (2017) http://dx.doi.org/10.1007/s40304-017-0117-6.
[20] H. Antil, R. Khatri, R. Löhner, D. Verma, Fractional deep neural network via constrained optimization, Mach. Learn.: Sci. Technol. 2 (1) (2020) http://dx.doi.org/10.1088/2632-2153/aba8e7.
[21] G. Pang, L. Lu, G.E. Karniadakis, FPINNs: Fractional physics-informed neural networks, SIAM J. Sci. Comput. 41 (4) (2019) A2603–A2626, http://dx.doi.org/10.1137/18M1229845.
[22] M. Raissi, P. Perdikaris, G.E. Karniadakis, Physics informed deep learning (Part I): Data-driven solutions of nonlinear partial differential equations, 2017, arXiv:1711.10561.
[23] C. Huré, H. Pham, X. Warin, Some machine learning schemes for high-dimensional nonlinear PDEs, Math. Comp. 89 (2020) 1547–1579.
[24] S. Wang, X. Yu, P. Perdikaris, When and why PINNs fail to train: A neural tangent kernel perspective, 2020, arXiv:2007.14527.
[25] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F.A. Hamprecht, Y. Bengio, A. Courville, On the spectral bias of neural networks, 2019, arXiv:1806.08734.
[26] S. Mishra, R. Molinaro, Estimates on the generalization error of physics informed neural networks (PINNs) for approximating PDEs, 2020, arXiv preprint arXiv:2006.16144.
[27] E. Kharazmi, Z. Zhang, G.E. Karniadakis, VPINNS: Variational physics-informed neural networks for solving partial differential equations, 2019, arXiv:1912.00873.
[28] X. Ying, An overview of overfitting and its solutions, J. Phys. Conf. Series 1168 (2019) 022022, http://dx.doi.org/10.1088/1742-6596/1168/2/022022.
[29] C. Zhang, O. Vinyals, R. Munos, S. Bengio, A study on overfitting in deep reinforcement learning, 2018, arXiv:1804.06893.
[30] E. Weinan, Y. Bing, The deep ritz method: A deep learning-based numerical algorithm for solving variational problems, Commun. Math. Stat. 6 (2018) 1–12, http://dx.doi.org/10.1007/s40304-018-0127-z.
[31] J. Sirignano, K. Spiliopoulos, DGM: A Deep learning algorithm for solving partial differential equations, J. Comput. Phys. 375 (2018) 1339–1364, http://dx.doi.org/10.1016/j.jcp.2018.08.029.
[32] Z. Mao, A.D. Jagtap, G.E. Karniadakis, Physics-informed neural networks for high-speed flows, Comput. Methods Appl. Mech. Eng. 360 (2020) 112789, http://dx.doi.org/10.1016/j.cma.2019.112789.
[33] M. Raissi, P. Perdikaris, G. 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, http://dx.doi.org/10.1016/j.jcp.2018.10.045.
[34] K. Liu, Automatic integration, 2020, arXiv:2006.15210.
[35] A. Güneş Baydin, B.A. Pearlmutter, A. Andreyevich Radul, J. Mark Siskind, Automatic differentiation in machine learning: A survey, J. Mach. Learn. Res. 18 (2018) 1–43, arXiv:1502.05767.
[36] Z. Cai, J. Chen, M. Liu, X. Liu, Deep least-squares methods: An unsupervised learning-based numerical method for solving elliptic PDEs, J. Comput. Phys. 420 (2020) 109707, http://dx.doi.org/10.1016/j.jcp.2020.109707.
[37] R. Khodayi-Mehr, M. Zavlanos, VarNet: VAriational neural networks for the solution of partial differential equations, in: A.M. Bayen, A. Jadbabaie, G. Pappas, P.A. Parrilo, B. Recht, C. Tomlin, M. Zeilinger (Eds.), Proceedings Of The 2nd Conference On Learning For Dynamics And Control, in: Proceedings of Machine Learning Research, 120, PMLR, The Cloud, 2020, pp. 298–307.
[38] W. Ritz, Über eine neue Methode zur Lösung gewisser Variationsprobleme der mathematischen Physik, J. Für Die Reine Und Angew. Math. 135 (1909) 1–61.
[39] D. Mortari, Least-squares solution of linear differential equations, Mathematics 5 (2017) http://dx.doi.org/10.3390/math5040048.
[40] C. Johnson, Numerical Solution Of Partial Differential Equations By The Finite Element Method, in: Dover Books on Mathematics Series, Dover Publications, Incorporated, 2012.
[41] I. Lagaris, A. Likas, D. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Netw. 9 (1998) 987–1000, http://dx.doi.org/10.1109/72.712178.
[42] P. Moin, Fundamentals Of Engineering Numerical Analysis, second ed., Cambridge University Press, 2010, http://dx.doi.org/10.1017/CBO9780511781438.
[43] M. Pharr, G. Humphreys (Eds.), 14 - Monte Carlo Integration I: Basic concepts, in: Physically Based Rendering, Morgan Kaufmann, Burlington, 2004, pp. 631–660, http://dx.doi.org/10.1016/B978-012553180-1/50016-8.
[44] S. Weinzierl, Introduction to Monte Carlo methods, ISBN: 978-0-387-87836-2, 2000, http://dx.doi.org/10.1007/978-0-387-87837-9_1, ArXiv High Energy Physics - Phenomenology E-Prints.
[45] F.B. Barros, S.P.B. Proença, C.S. de Barcellos, On error estimator and p-adaptivity in the generalized finite element method, Int. J. Num. Methods Eng. 60 (14) (2004) 2373–2398, http://dx.doi.org/10.1002/nme.1048.
[46] L. Demkowicz, W. Rachowicz, P. Devloo, A fully automatic hp-adaptivity, J. Sci. Comput. 17 (2002) http://dx.doi.org/10.1023/A:1015192312705.
[47] M. Fazlyab, A. Robey, H. Hassani, M. Morari, G. Pappas, Efficient and accurate estimation of Lipschitz constants for deep neural networks, Adv. Neural Inf. Process. Syst. 32 (2019) 11427–11438.

[48] H. Gouk, E. Frank, B. Pfahringer, M.J. Cree, Regularisation of neural networks by enforcing lipschitz continuity, Mach. Learn. 110 (2) (2021) 393–416.
[49] K. Scaman, A. Virmaux, Lipschitz regularity of deep neural networks: analysis and efficient estimation, in: Proceedings Of The 32nd International Conference On Neural Information Processing Systems, 2018, pp. 3839–3848.
[50] V. Ruas, An introduction to the mathematical foundations of the finite element method, 2002.
[51] C.J. Budd, W. Huang, R.D. Russell, Adaptivity with moving grids, Acta Numer. 18 (2009) 111–241, http://dx.doi.org/10.1017/S0962492906400015.