ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/321160737

A unified deep artificial neural network approach to partial differential equations in complex geometries

Article *in* Neurocomputing · November 2017

DOI: 10.1016/j.neucom.2018.06.056

CITATIONS
305

READS
870

2 authors:

Jens Berg

18 PUBLICATIONS **606** CITATIONS

SEE PROFILE

Kaj Nyström

Uppsala University

117 PUBLICATIONS **1,539** CITATIONS

SEE PROFILE

All content following this page was uploaded by Jens Berg on 04 December 2017.

The user has requested enhancement of the downloaded file.

arXiv:1711.06464v1 [stat.ML] 17 Nov 2017

A unified deep artificial neural network approach to partial differential equations in complex geometries

Jens Berg* and Kaj Nyström†

Department of Mathematics, Uppsala University
S-751 06 Uppsala, Sweden

Abstract

We use deep feedforward artificial neural networks to approximate solutions of partial differential equations of advection and diffusion type in complex geometries. We derive analytical expressions of the gradients of the cost function with respect to the network parameters, as well as the gradient of the network itself with respect to the input, for arbitrarily deep networks. The method is based on an ansatz for the solution, which requires nothing but feedforward neural networks, and an unconstrained gradient based optimization method such as gradient descent or quasi-Newton methods.

We provide detailed examples on how to use deep feedforward neural networks as a basis for further work on deep neural network approximations to partial differential equations. We highlight the benefits of deep compared to shallow neural networks and other convergence enhancing techniques.

1 Introduction

Partial differential equations (PDEs) are used to model a variety of phenomena in the natural sciences. Common to most of the PDEs encountered in practical applications is that they cannot be solved analytically, but require various approximation techniques. Traditionally, mesh based methods such as finite elements (FEM), finite

*jens.berg@math.uu.se
†kaj.nystrom@math.uu.se

differences (FDM), or finite volumes (FVM), are the dominant techniques for obtaining approximate solutions. These techniques require that the computational domain of interest is discretized into a set of mesh points, and the solution is approximated at the points of the mesh. The advantage of these methods is that they are very efficient for low-dimensional problems on regular geometries. The drawback is that for complicated geometries, meshing can be as difficult as the numerical solution of the PDE itself. Moreover, the solution is only computed in the mesh points and evaluation of the solution in any other point requires interpolation or some other reconstruction method.

Other methods do not require a mesh but a set of collocation points where the solution is approximated. The collocation points can be generated according to some distribution inside the domain of interest and examples include radial basis functions (RBF) and Monte Carlo methods (MCM). The advantage is that it is relatively easy to generate collocation points inside the domain by a hit-and-miss approach. The drawbacks of these methods, compared to traditional mesh based methods, are for example numerical stability for RBF and inefficiency for MCM.

The last decade has seen a revolution in deep learning, where deep artificial neural networks (ANNs) are the key component. ANNs is not a new concept. They have been around since the 40's [25] and used for various applications. A historical review, in particular in the context of differential equations can be found in ch. 2 of [36]. The success of deep learning during the last decade is due to a combination of improved theory, starting with unsupervised pre-training and deep belief nets, and improved hardware resources such as general purpose graphics processing units (GPGPUs). See for example [11, 20]. Deep ANNs are now routinely used with impressive results in areas ranging from image analysis, pattern recognition, object detection, natural language processing, to self-driving cars. Simple feedforward ANNs have not yet enjoyed the same success as recurrent ANNs, for natural language processing, or convolutional ANNs, for image analysis. However, recent results show that feedforward ANNs have great potential by allowing self-normalizing activation functions [18].

While deep ANNs have achieved impressive results in several important application, there are still many open questions concerning how and why they actually work so efficiently. From the perspective of function approximation theory it has been known since the 90's that ANNs are universal approximators that can be used to approximate any continuous function and its derivatives [12, 13, 5, 24]. In the context of PDEs, single hidden layer ANNs have traditionally been used to solve PDEs since one hidden layer with sufficiently many neurons is sufficient for approximating any function, and all gradients that are needed can be computed in analytical closed form [22, 23, 26]. More recently, there is a limited but emerging literature on

the use of deep ANNs to solve PDEs [31, 30, 6, 3]. For an overview of some of the developments we refer the reader to [21]. In general ANNs have the benefits that they are smooth, analytical functions which can be evaluated at any point inside, or even outside, the domain without reconstruction.

The aim of this paper is twofold. The first is to introduce a unified method of solving PDEs which only involves feedforward deep ANNs with (close to) no user intervention. In previous works, see for example Lagaris et. al. [22], only a single hidden layer is used, and the method outlined requires some user-defined functions which are non-trivial, or even impossible, to compute. Later, Lagaris et. al [23] removed the need for user intervention by using a combination of feedforward and radial basis function ANNs in a non-trivial way. Later work by McFall and Mahan [26] removed the need for the radial basis function ANN by replacing it with length factors that are computed using thin plate splines. The thin plate splines computation is, however, rather cumbersome as it involves the solution of many linear systems which adds extra complexity to the problem. In contrast, the method proposed in this paper requires only an implementation of a deep feedforward ANN together with a cost function and any choice of unconstrained numerical optimization method. We derive analytical expressions for the gradient of the cost function with respect to the network parameters, together with the gradients of the network with respect to the input, for arbitrarily deep networks. This is the focus in the first part of the paper.

The second aim is to propose PDEs as a vehicle for further theoretical explorations of deep ANNs. Most applications of deep ANNs, in for example image analysis or audio processing, are too complicated to be used as analytical tools. In contrast, with PDEs we can have complete understanding of the input-output relations and we can study the training process for various ANN configurations. This is the focus in the second part of the paper. In this context we recall that a general problem in deep learning is to find a sufficient amount of labeled data to use in training. While this is becoming less and less of a problem due to the community's effort to produce open datasets, we simply note that by using the PDE itself we can generate as much data as needed.

The rest of the paper is organized as follows. In section 2 we recall the network architecture of deep, fully connected feedforward ANNs, we introduce some necessary notation and the backpropagation scheme. In section 3 we develop unified artificial neural network approximations for stationary PDEs. Based on our ansatz for the solution we discuss extension of the boundary data, smooth distance function approximation and computation, gradient computations and the backpropagation algorithm for parameter calibration. We give detailed account of the latter compu-

tations/algorithms in the case of advection and diffusion problems. In section 4 we provide some concrete examples concerning how to apply the method outlined. Our examples include linear advection and diffusion in 1D and 2D, but high-dimensional problems are also discussed. Section 5 is devoted to convergence considerations. As training neural networks consumes a lot of time and computational resources, it is desirable to reduce the number of required iterations until a certain accuracy has been reached. During this work, we have found two factors which strongly influence the number of required iterations in our PDE focused applications/examples. The first is pre-training the network using the available boundary data, and the second is to increase the number of hidden layers. In section 5 we discuss this in more detail and one particular finding is that we, by fixing the capacity of the network and increasing the number of hidden layers, can see a dramatic decrease in the number of iterations required to reach a desired level of accuracy. Our findings indicate that using deep ANNs instead of just shallow ones adds real value in the context of using ANNs to solve PDEs. Finally, section 6 is devoted to a summary, conclusions and an outlook on future work.

2 Network architecture

In this paper, we consider deep, fully connected feedforward ANNs. The ANN consists of $L+1$ layers, where layer 0 is the input layer and layer L is the output layer. The layers $0 < l < L$ are the hidden layers. The activation functions in the hidden layers can be any activation function such as for example sigmoids, rectified linear units, or hyperbolic tangents. Unless otherwise stated, we will use sigmoids in the hidden layers. The output activation will be the linear activation. The ANN defines a mapping $\mathbb{R}^N \to \mathbb{R}^M$.

Each neuron in the ANN is supplied with a bias, including the output neurons but excluding the input neurons, and the connections between neurons in subsequent layers are represented by matrices of weights. We let b_j^l denote the bias of neuron j in layer l. The weight between neuron k in layer $l-1$, and neuron j in layer l is denoted by w_{jk}^l. The activation function in layer l will be denoted by σ_l regardless of the type of activation. We assume for simplicity that a single activation function is used for each layer. The output from neuron j in layer l will be denoted by y_j^l. See Figure 1 for a schematic representation of a fully connected feedforward ANN.

A quantity that will be extensively used is the so-called weighted input which is defined as

$$z_j^l = \sum_k w_{jk}^l \sigma_{l-1}(z_k^{l-1}) + b_j^l, \tag{2.1}$$

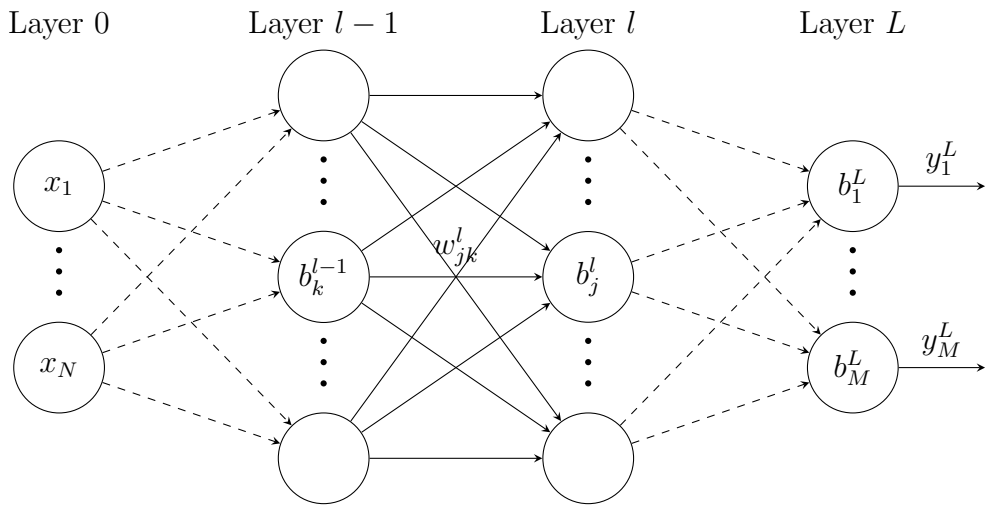


Figure 1: Schematic representation of a fully connected feedforward ANN.

where the sum is taken over all inputs to neuron j in layer l. That is, the number of neurons in layer $l-1$. The weighted input (2.1) can of course also be written in terms of the output from the previous layer as

$$z_j^l = \sum_k w_{jk}^l y_k^{l-1} + b_j^l, \tag{2.2}$$

where the output $y_k^{l-1} = \sigma_{l-1}(z_k^{l-1})$ is the activation of the weighted input. As we will be working with deep ANNs, we will prefer formula (2.1) as it naturally defines a recursion in terms of previous weighted inputs through the ANN. By definition we have

$$\sigma_0(z_j^0) = y_j^0 = x_j, \tag{2.3}$$

which terminates any recursion.

By dropping the subscripts we can write (2.1) in the convenient vectorial form

$$z^l = W^l \sigma_{l-1}(z^{l-1}) + b^l = W^l y^{l-1} + b^l, \tag{2.4}$$

where each element in the z^l and y^l vectors are given by z_j^l and y_j^l, respectively, and the activation function is applied elementwise. The elements of the matrix W^l are given by $W_{jk}^l = w_{jk}^l$.

With the above definitions, the feedforward algorithm for computing the output

y^L, given the input x, is given by

$$\begin{aligned} y^L &= \sigma_L(z^L) \\ z^L &= W^L\sigma_{L-1}(z^{L-1}) + b^L \\ z^{L-1} &= W^{L-1}\sigma_{L-2}(z^{L-2}) + b^{L-1} \\ &\vdots \\ z^2 &= W^2\sigma_1(z^1) + b^2 \\ z^1 &= W^1x + b^1. \end{aligned} \tag{2.5}$$

2.1 Backpropagation

Given some data $x = [x_1, \ldots, x_N]^T \in \mathbb{R}^N$ and some target outputs $y = [y_1, \ldots, y_M]^T \in \mathbb{R}^M$ we wish to choose our weights and biases such that $y^L(x; w, b)$ is a good approximation of $y(x)$. We use the notation $y^L(x; w, b)$ to indicate that the ANN takes x as input and that the ANN is parametrized by the weights and biases, w, b. To find the weights and biases, we define some cost function $C = C(y, y^L) : \mathbb{R}^N \to \mathbb{R}$ and compute

$$w^*, b^* = \arg\min_{w,b} C(y, y^L). \tag{2.6}$$

The minimization problem (2.6) can be solved using a variety of optimization methods, both gradient based and gradient free. In this paper, we will focus on gradient based methods and derive the gradients that we need in order to use any gradient based optimization method. The standard way of computing gradients is by using backpropagation. The main ingredient is the error of neuron j in layer l defined by

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}. \tag{2.7}$$

The standard backpropagation algorithm for computing the gradients of the cost function is then given by

$$\begin{aligned} \delta_j^L &= \frac{\partial C}{\partial y_j^L}\sigma_L'(z_j^L), & \frac{\partial C}{\partial w_{jk}^l} &= y_k^{l-1}\delta_j^l, \\ \delta_j^l &= \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma_l'(z_j^l), & \frac{\partial C}{\partial b_j^l} &= \delta_j^l. \end{aligned} \tag{2.8}$$

The δ-terms in (2.8) can be written in vectorial form as

$$\delta^L = \nabla_{y^L} C \odot \sigma_L'(z^L), \quad \delta^l = (W^{l+1})^T\delta^{l+1} \odot \sigma_l'(z^l), \tag{2.9}$$

where we use the notation $\nabla_{y^L} C = [\frac{\partial C}{\partial y_1^L}, \ldots, \frac{\partial C}{\partial y_M^L}]^T$, and $\odot$ denotes the Hadamard, or componentwise, product. The notation ∇C, without a subscript, will denote the vector of partial derivatives with respect to the inputs, x. Note the transpose on the weight matrix W. The transpose W^T is the Jacobian matrix of the coordinate transformation between layer $l+1$ and l.

3 Unified ANN approximations to PDEs

In this section we interested in solving stationary PDEs of the form

$$\begin{aligned} Lu &= f, \quad x \in \Omega, \\ Bu &= g, \quad x \in \Gamma \subset \partial\Omega, \end{aligned} \tag{3.1}$$

where L is a differential operator, f a forcing function, B a boundary operator, and g the boundary data. The domain of interest is $\Omega \subset \mathbb{R}^N$, $\partial\Omega$ denotes its boundary, and Γ is the part of the boundary where boundary conditions should be imposed. In this paper, we focus on problems where L is either the advection or diffusion operator, or some mix thereof. The extension to higher order problems are conceptually straightforward based on the derivations outlined below.

We consider the ansatz $\hat{u} = \hat{u}(x; w, b)$ for u where (w, b) denotes the parameters of the underlying ANN. To determine the parameters we will use the cost function defined as the quadratic residual,

$$C = \frac{1}{2}||L\hat{u} - f||^2 := \frac{1}{2}\int_\Omega |L\hat{u} - f|^2 \, dx. \tag{3.2}$$

For further reference we need the gradient of (3.2) with respect to any network parameter. Let p denote any of w_{jk}^l or b_j^l. Then

$$\frac{\partial C}{\partial p} = (L\hat{u} - f) \odot (L\frac{\partial \hat{u}}{\partial p}). \tag{3.3}$$

We will use the collocation method [22] and therefore we discretize Ω and Γ into a sets of collocation points Ω_d and Γ_d, with $|\Omega_d| = N_d$ and $|\Gamma_d| = N_b$, respectively. In its discrete form the minimization problem (2.6) then becomes

$$w^*, b^* = \arg\min_{w,b} \sum_{x_i \in \Omega_d} \frac{1}{2}\frac{1}{N_d}||L\hat{u}(x_i) - f(x_i)||^2, \tag{3.4}$$

subject to the constraints

$$B\hat{u}(x_i) = g(x_i), \quad \forall x_i \in \Gamma_d. \tag{3.5}$$

There are many approaches to the minimization problem (3.4) subject to the constraints (3.5). One can for example use a constraint optimization procedure, Lagrange multipliers, or penalty formulations to include the constraints in the optimization. Another approach is to design the ansatz $\hat{u}$, such that the constraints are automatically fulfilled, thus turning the constrained optimization problem into an unconstrained optimization problem. Unconstrained optimization problems can be more efficiently solved using gradient based optimization techniques.

In the following, we let, for simplicity, B in (3.1) be the identity operator, and we hence consider PDEs with Dirichlet boundary conditions. The specific ansatz for the solution we consider in the following is

$$\hat{u}(x) = G(x) + D(x)y^L(x; w, b), \tag{3.6}$$

where $G = G(x)$ is a smooth extension of the boundary data g and $D = D(x)$ is a smooth distance function giving the distance for $x \in \Omega$ to Γ. Note that the form of the ansatz (3.6) ensures that $\hat{u}$ attains its boundary values at the boundary points. Moreover, G and D are pre-computed using low-capacity ANNs for the given boundary data and geometry using only a small subset of the collocation points, and do not add any extra complexity when minimizing (3.4). We call this approach unified as there are no other ingredients involved other than feedforward ANNs, and gradient based, unconstrained optimization methods to compute all of G, D, and y^L.

3.1 Extension of the boundary data

The ansatz (3.6) requires that G is globally defined, smooth[1], and that

$$|G(x) - g(x)| < \epsilon, \quad \forall x \in \Gamma. \tag{3.7}$$

Other than these requirements, the exact form of G is not important. It is hence an ideal candidate for an ANN approximation. To compute G, we simply train an ANN to fit $g(x)$, $\forall x \in \Gamma_d$. The quadratic cost function used is given by

$$C = \frac{1}{2}\frac{1}{N_b}\sum_{x_i \in \Gamma_d} ||G(x_i) - g(x_i)||^2, \tag{3.8}$$

[1]It is sufficient that G has continuous derivatives up to the order of the differential operator L. However, since G will be computed using an ANN, it will be smooth.

and in the course of calibration we compute the gradients using the standard back-propagation (2.8). Note that in general we have $N_b << N_d$ and the time it takes to compute G is of lower order.

3.2 Smooth distance function computation

To compute the smooth distance function D, we start by computing a non-smooth distance function, d, and approximate it using a low-capacity ANN. For each point, x, we define d as the minimum distance to a boundary point where a boundary condition should be imposed. That is,

$$d(x) = \min_{x_b \in \Gamma} ||x - x_b||. \tag{3.9}$$

Again, the exact form of d (and D) is not important other than that D is smooth and

$$|D(x)| < \epsilon, \quad \forall x \in \Gamma. \tag{3.10}$$

We can use a small subset $\omega_d \subset \Omega_d$, with $|\omega_d| = n_d << N_d$ to compute d.

Once d has been computed and normalized, we fit another ANN and train it using the cost function

$$C = \frac{1}{2} \frac{1}{n_d + N_b} \sum_{x_i \in \omega_d \cup \Gamma_d} ||D(x_i) - d(x_i)||^2. \tag{3.11}$$

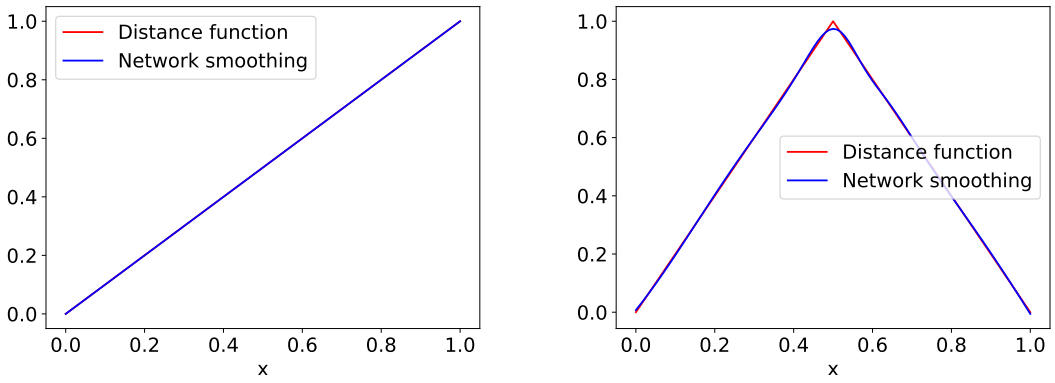
Note that we train the ANN over the points $x_i \in \omega_d \cup \Gamma_d$, with $d(x_i) = 0 \ \forall x_i \in \Gamma_d$. Since the ANN defining the smooth distance function can be taken to be a single hidden layer, low-capacity ANN with $n_d + N_b << N_d$, the total time to compute both d and D is of lower order. See Figure 2 for (rather trivial examples) of d and D in one dimension.

Remark 3.1. We stress that the distance function (3.9) can be computed efficiently with nearest-neighbor searches, using for example k-d trees [1], or ball trees [29] for very high-dimensional input. A naïve nearest neighbor search will have $O(N_d N_b)$ complexity, while an efficient nearest-neighbor search will typically have $O(N_d \log N_b)$.

Remark 3.2. Instead of computing the actual distance function, we could use the more extreme version

$$d(x) = \begin{cases} 0, & x \in \Gamma, \\ 1, & \text{otherwise.} \end{cases} \tag{3.12}$$

However, simulations showed that this function and its approximating ANN have a negative impact on convergence and quality when computing $\hat{u}$.



(a) $L = d/dx$. Only one boundary condition at $x = 0$.

(b) $L = d^2/dx^2$. Boundary conditions at both boundary points.

Figure 2: Smoothed distance functions using single hidden layer ANNs with 5 hidden neurons, using 100 collocation points.

3.3 Gradient computations

When the differential operator L acts on the ansatz $\hat{u}$ in (3.6), we need, as $\Omega \subset \mathbb{R}^N$, to compute the partial derivatives of the ANNs with respect to the spatial variables $(x_1, ..., x_N)$. Also, when using gradient based optimization we need to compute the gradients of the quadratic residual cost function (3.2), and also the ANNs, with respect to the network parameters.

In the single hidden layer case, all gradients can be computed in closed analytical form as shown in [22]. For deep ANNs, however, we need to modify the feedforward (2.5) and backpropagation (2.8) algorithms to compute the gradients with respect to $(x_1, ..., x_N)$ and network parameters, respectively.

Note that the minimization problem in (3.4) involves the collocation points $\{x_i\}$. In the following, and throughout the paper, we will, with a slight abuse of notation, by $\frac{\partial}{\partial x_i}$ always denote the derivative with respect to the spatial coordinate x_i, not with respect to collocation point x_i.

3.3.1 Advection problems

For advection problems we have a PDE of the form

$$Lu = \nabla \cdot (Vu) = f, \tag{3.13}$$

for some (non-linear) matrix coefficient V. When the advection operator L acts on $\hat{u}$ we need to compute the gradient of the ANN with respect to the input. The gradients can be computed by taking partial derivatives of (2.5). We get, in component form,

$$\begin{aligned}
\frac{\partial y_j^L}{\partial x_i} &= \sigma'(z_j^L)\frac{\partial z_j^L}{\partial x_i} \\
\frac{\partial z_j^L}{\partial x_i} &= \sum_k w_{jk}^L \sigma'_{L-1}(z_k^{L-1})\frac{\partial z_k^{L-1}}{\partial x_i} \\
\frac{\partial z_j^{L-1}}{\partial x_i} &= \sum_k w_{jk}^{L-1} \sigma'_{L-2}(z_k^{L-2})\frac{\partial z_k^{L-2}}{\partial x_i} \\
&\vdots \\
\frac{\partial z_j^2}{\partial x_i} &= \sum_k w_{jk}^2 \sigma'_1(z_k^1)\frac{\partial z_k^1}{\partial x_i} \\
\frac{\partial z_j^1}{\partial x_i} &= w_{ji}^1.
\end{aligned} \tag{3.14}$$

Note the slight abuse of subscript notation in order to avoid having too many subscripts. Note also that (3.14) defines a new ANN with the same weights, but no biases, as the original ANN with modified activation functions. A convenient vectorial form is obtained by dropping the subscripts and re-writing (3.14) as

$$\begin{aligned}
\frac{\partial y^L}{\partial x_i} &= \sigma'_L(z^L) \odot \frac{\partial z^L}{\partial x_i} \\
\frac{\partial z^L}{\partial x_i} &= W^L \sigma'_{L-1}(z^{L-1}) \odot \frac{\partial z^{L-1}}{\partial x_i} \\
\frac{\partial z^{L-1}}{\partial x_i} &= W^{L-1} \sigma'_{L-2}(z^{L-2}) \odot \frac{\partial z^{L-2}}{\partial x_i} \\
&\vdots \\
\frac{\partial z^2}{\partial x_i} &= W^2 \sigma'_1(z^1) \odot \frac{\partial z^1}{\partial x_i} \\
\frac{\partial z^1}{\partial x_i} &= W^1 e_i,
\end{aligned} \tag{3.15}$$

where e_i is the ith standard basis vector. To compute all gradients simultaneously, we define, for a vector v and vector-valued function f, the diagonal and Jacobian

matrices

$$\Sigma(v) = \begin{bmatrix} v_1 & 0 & \cdots & 0 \\ 0 & v_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_N \end{bmatrix}, \quad J(f) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_{M-1}}{\partial x_1} & \cdots & \frac{\partial f_{M-1}}{\partial x_{N-1}} & \frac{\partial f_{M-1}}{\partial x_N} \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_{N-1}} & \frac{\partial f_M}{\partial x_N} \end{bmatrix}. \tag{3.16}$$

Using these matrices, we can write all partial derivatives in the compact matrix form

$$\begin{aligned} J(y^L) &= \Sigma(\sigma_L'(z^L))J(z^L) \\ J(z^L) &= W^L\Sigma(\sigma_{L-1}'(z^{L-1}))J(z^{L-1}) \\ J(z^{L-1}) &= W^{L-1}\Sigma(\sigma_{L-2}'(z^{L-2}))J(z^{L-2}) \\ &\vdots \\ J(z^2) &= W^2\Sigma(\sigma_1'(z^1))J(z^1) \\ J(z^1) &= W^1 I_{N\times N}, \end{aligned} \tag{3.17}$$

where $I_{N\times N}$ is the $N \times N$ identity matrix. All gradients of the ANN with respect to the input are contained in the rows of the Jacobian matrix $J(y^L)$ as

$$J(y^L) = \begin{bmatrix} \cdots & (\nabla y_1^L)^T & \cdots \\ \cdots & \vdots & \cdots \\ \cdots & (\nabla y_M^L)^T & \cdots \end{bmatrix}. \tag{3.18}$$

To solve the PDE (3.13), we need to modify the backpropagation algorithm (2.8) to include the gradients of the residual cost function (3.2) with respect to the network parameters. First, we need to compute

$$\frac{\partial y_m^L}{\partial w_{jk}^l}, \quad \frac{\partial y_m^L}{\partial b_j^l}, \quad m = 1, \ldots, M. \tag{3.19}$$

This can be done componentwise by using backpropagation with the identity as the cost function. That is, we let $C(y_m^L) = y_m^L$ and then (2.8) becomes

$$\begin{aligned} \delta_m^L &= \sigma_L'(z_m^L), & \frac{\partial y_m^L}{\partial w_{jk}^l} &= y_k^{l-1}\delta_j^l, \\ \delta_j^l &= \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma_l'(z_j^l), & \frac{\partial y_m^L}{\partial b_j^l} &= \delta_j^l. \end{aligned} \tag{3.20}$$

The δ-terms in (3.20) can be written in vectorial form as

$$\begin{aligned} \delta^L &= \sigma_L'(z^L), \\ \delta^l &= (W^{l+1})^T \delta^{l+1} \odot \sigma_l'(z^l). \end{aligned} \tag{3.21}$$

Secondly, we need to compute

$$\frac{\partial^2 y_m^L}{\partial x_i \partial w_{jk}^l}, \quad \frac{\partial^2 y_m^L}{\partial x_i \partial b_j^l}, \quad i = 1, \ldots, N, \quad m = 1, \ldots, M. \tag{3.22}$$

This can be done by taking the partial derivative of (3.20) with respect to x_i. We get

$$\begin{aligned} \frac{\delta_m^L}{\partial x_i} &= \sigma_L''(z_m^L) \frac{\partial z_m^L}{\partial x_i}, \\ \frac{\partial \delta_j^l}{\partial x_i} &= \sum_k w_{kj}^{l+1} \left(\frac{\partial \delta_k^{l+1}}{\partial x_i} \sigma_l'(z_j^l) + \delta_k^{l+1} \sigma_l''(z_j^l) \frac{\partial z_j^l}{\partial x_i} \right), \\ \frac{\partial^2 y_m^L}{\partial x_i \partial w_{jk}^l} &= \frac{\partial y_k^{l-1}}{\partial x_i} \delta_j^l + y_k^{l-1} \frac{\partial \delta_j^l}{\partial x_i}, \\ \frac{\partial^2 y_m^L}{\partial x_i \partial b_j^l} &= \frac{\partial \delta_j^l}{\partial x_i}. \end{aligned} \tag{3.23}$$

The δ-terms in (3.23) can again be written in vectorial form as

$$\begin{aligned} \frac{\delta^L}{\partial x_i} &= \sigma_L''(z^L) \odot \frac{\partial z^L}{\partial x_i}, \\ \frac{\partial \delta^l}{\partial x_i} &= (W^{l+1})^T \left(\frac{\partial \delta^{l+1}}{\partial x_i} \odot \sigma_l'(z^l) + \delta^{l+1} \odot \sigma_l''(z^l) \odot \frac{\partial z^l}{\partial x_i} \right), \end{aligned} \tag{3.24}$$

or in matrix form as

$$\begin{aligned} J(\delta^L) &= \Sigma(\sigma_L''(z^L)) J(z^L), \\ J(\delta^l) &= (W^{l+1})^T \left(\Sigma(\sigma_l'(z^l)) J(\delta^{l+1}) + \Sigma(\sigma_l''(z^l)) \Sigma(\delta^{l+1}) J(z^l) \right). \end{aligned} \tag{3.25}$$

The feedforward and backpropagation algorithms (2.5), (3.14), (3.20), and (3.23), are sufficient to compute the solution to any first-order PDE, both linear and non-linear.

Remark 3.3. Note that all data from a standard feedforward and backpropagation pass is re-used in the computation of (3.14) and (3.23). The complexity of computing (3.14) and (3.23) is thus the same as when computing (2.5) and (2.8), with some additional memory overhead to store the activations and weighted inputs.

3.3.2 Diffusion problems

For diffusion problems we have a PDE of the form

$$Lu = \nabla \cdot (V \nabla u) = f. \tag{3.26}$$

When the diffusion operator acts on the ansatz, we get second-order terms which we have not already computed. We here first give the computations in the case of two derivatives with respect to x_i. Indeed, we take the partial derivative of (3.14) with respect to x_i and get

$$\begin{aligned}
\frac{\partial^2 y_m^L}{\partial x_i^2} &= \sigma_L''(z_m^L)\left(\frac{\partial z_m^L}{\partial x_i}\right)^2 + \sigma_L'(z_m^L)\frac{\partial^2 z_m^L}{\partial x_i^2} \\
\frac{\partial^2 z_j^L}{\partial x_i^2} &= \sum_k w_{jk}^L \left(\sigma_{L-1}''(z_k^{L-1})\left(\frac{\partial z_k^{L-1}}{\partial x_i}\right)^2 + \sigma_{L-1}'(z_k^{L-1})\frac{\partial^2 z_k^{L-1}}{\partial x_i^2}\right) \\
\frac{\partial^2 z_j^{L-1}}{\partial x_i^2} &= \sum_k w_{jk}^{L-1} \left(\sigma_{L-2}''(z_k^{L-2})\left(\frac{\partial z_k^{L-2}}{\partial x_i}\right)^2 + \sigma_{L-2}'(z_k^{L-2})\frac{\partial^2 z_k^{L-2}}{\partial x_i^2}\right) \\
&\vdots \\
\frac{\partial^2 z_j^2}{\partial x_i^2} &= \sum_k w_{jk}^2 \left(\sigma_1''(z_k^1)\left(\frac{\partial z_k^1}{\partial x_i}\right)^2 + \sigma_1'(z_k^1)\frac{\partial^2 z_k^1}{\partial x_i^2}\right) \\
\frac{\partial^2 z_j^1}{\partial x_i^2} &= 0.
\end{aligned} \tag{3.27}$$

The vectorial form of (3.27) can be written as

$$\begin{aligned}
\frac{\partial^2 y^L}{\partial x_i^2} &= \sigma_L''(z^L)\left(\frac{\partial z^L}{\partial x_i}\right)^2 + \sigma_L'(z^L)\frac{\partial^2 z^L}{\partial x_i^2} \\
\frac{\partial^2 z^L}{\partial x_i^2} &= W^L\left(\sigma_{L-1}''(z^{L-1}) \odot \left(\frac{\partial z^{L-1}}{\partial x_i}\right)^2 + \sigma_{L-1}'(z^{L-1}) \odot \frac{\partial^2 z^{L-1}}{\partial x_i^2}\right) \\
\frac{\partial^2 z^{L-1}}{\partial x_i^2} &= W^{L-1}\left(\sigma_{L-2}''(z^{L-2}) \odot \left(\frac{\partial z^{L-2}}{\partial x_i}\right)^2 + \sigma_{L-2}'(z^{L-2}) \odot \frac{\partial^2 z^{L-2}}{\partial x_i^2}\right) \\
&\vdots \\
\frac{\partial^2 z^2}{\partial x_i^2} &= W^2\left(\sigma_1''(z^1) \odot \left(\frac{\partial z^1}{\partial x_i}\right)^2 + \sigma_1'(z^1) \odot \frac{\partial^2 z^1}{\partial x_i^2}\right) \\
\frac{\partial^2 z^1}{\partial x_i^2} &= 0_N,
\end{aligned} \tag{3.28}$$

where $0_N = [0, \ldots, 0]^T$ denotes the zero vector with N elements. To write the matrix form of (3.27) we define $J^2(f)$ to be the matrix of non-mixed second partial derivatives of a vector-valued function f,

$$J^2(f) = \begin{bmatrix}
\frac{\partial^2 f_1}{\partial x_1^2} & \frac{\partial^2 f_1}{\partial x_2^2} & \cdots & \frac{\partial^2 f_1}{\partial x_N^2} \\
\frac{\partial^2 f_2}{\partial x_1^2} & \frac{\partial^2 f_2}{\partial x_2^2} & \cdots & \frac{\partial^2 f_2}{\partial x_N^2} \\
\vdots & \vdots & \vdots & \vdots \\
\frac{\partial^2 f_{M-1}^2}{\partial x_1} & \cdots & \frac{\partial^2 f_{M-1}}{\partial x_{N-1}^2} & \frac{\partial^2 f_{M-1}}{\partial x_N^2} \\
\frac{\partial^2 f_M}{\partial x_1^2} & \cdots & \frac{\partial^2 f_M}{\partial x_{N-1}^2} & \frac{\partial^2 f_M}{\partial x_N^2}
\end{bmatrix}. \tag{3.29}$$

The matrix form can then be written as

$$\begin{aligned}
J^2(y^L) &= \Sigma(\sigma''(z^L))J(z^L)^2 + \Sigma(\sigma_L'(z^L))J^2(z^L) \\
J^2(z^L) &= W^L(\Sigma(\sigma_{L-1}''(z^{L-1}))J(z^{L-1})^2 + \Sigma(\sigma_{L-1}'(z^{L-1}))J^2(z^{L-1})) \\
J^2(z^{L-1}) &= W^{L-1}(\Sigma(\sigma_{L-2}''(z^{L-2}))J(z^{L-2})^2 + \Sigma(\sigma_{L-2}'(z^{L-2}))J^2(z^{L-2})) \\
&\vdots \\
J^2(z^2) &= W^2(\Sigma(\sigma_1''(z^1))J(z^1)^2 + \Sigma(\sigma_1'(z^1))J^2(z^1)) \\
J^2(z^1) &= W^1 Z_{N\times N},
\end{aligned} \tag{3.30}$$

where $Z_{N\times N}$ is the zero matrix of dimension $N \times N$.

The gradient of the residual cost function (3.3) requires that we compute the third-order terms

$$\frac{\partial^3 y_m^L}{\partial x_i^2 \partial w_{jk}^l}, \quad \frac{\partial^3 y_m^L}{\partial x_i^2 \partial b_j^l}, \quad i = 1, \ldots, N, \quad m = i, \ldots, M. \tag{3.31}$$

These can be computed by taking the partial derivative of (3.23) with respect to x_i to get

$$\begin{aligned}
\frac{\partial^2 \delta_j^L}{\partial x_i^2} &= \sigma_L^{(3)}(z_j^L)\frac{\partial z_j^L}{\partial x_i} + \sigma_L''(z_j^L)\frac{\partial^2 z_j^L}{\partial x_i^2}, \\
\frac{\partial^2 \delta_j^l}{\partial x_i^2} &= \sum_k w_{jk}^{l+1}\left(\frac{\partial^2 \delta_k^{l+1}}{\partial x_i^2}\sigma_l'(z_j^l) + 2\frac{\partial \delta_k^{l+1}}{\partial x_i}\sigma_l''(z_j^l)\frac{\partial z_j^l}{\partial x_i}\right) \\
&\quad + \sum_k w_{jk}^{l+1}\left(\delta_k^{l+1}\sigma_l^{(3)}(z_j^l)\left(\frac{\partial z_j^l}{\partial x_i}\right)^2 + \delta_k^{l+1}\sigma_k''(z_l^j)\frac{\partial z_j^l}{\partial x_i}\right), \\
\frac{\partial^3 y_m^L}{\partial x_i^2 w_{jk}^l} &= \frac{\partial^2 y_k^{l-1}}{\partial x_i^2}\delta_j^l + 2\frac{\partial y_k^{l-1}}{\partial x_i}\frac{\partial \delta_j^l}{\partial x_i} + y_k^{l-1}\frac{\partial^2 \delta_k^{l-1}}{\partial x_i^2}, \\
\frac{\partial^3 y_m^L}{\partial x_i^2 b_j^l} &= \frac{\partial^2 \delta_j^l}{\partial x_i^2}.
\end{aligned} \tag{3.32}$$

The vectorial form of the δ-terms in (3.32) can be written as

$$\begin{aligned}
\frac{\partial^2 \delta^L}{\partial x_i^2} &= \sigma_L^{(3)}(z^L) \odot \frac{\partial z^L}{\partial x_i} + \sigma_L''(z^L) \odot \frac{\partial^2 z^L}{\partial x_i^2}, \\
\frac{\partial^2 \delta^l}{\partial x_i^2} &= (W^{l+1})^T\left(\frac{\partial^2 \delta^{l+1}}{\partial x_i^2} \odot \sigma_l'(z^l) + 2\frac{\partial \delta^{l+1}}{\partial x_i} \odot \sigma_l''(z^l) \odot \frac{\partial z^l}{\partial x_i}\right) \\
&\quad + (W^{l+1})^T\left(\delta^{l+1} \odot \sigma_l^{(3)}(z^l) \odot \left(\frac{\partial z^l}{\partial x_i}\right)^2 + \delta^{l+1} \odot \sigma_l''(z^l) \odot \frac{\partial z^l}{\partial x_i}\right),
\end{aligned} \tag{3.33}$$

and the matrix form as

$$\begin{aligned}
J^2(\delta^L) &= \Sigma(\sigma^{(3)_L}(z^L))J(z^L) + \Sigma(\sigma_L''(z^L))J^2(z^L), \\
J^2(\delta^l) &= (W^{l+1})^T\left(\Sigma(\sigma_l'(z^l))J^2(\delta^{l+1}) - 2\Sigma(\sigma_l''(z^l))J(\delta^{l+1})J(z^l)\right) \\
&\quad + (W^{l+1})^T\left(\Sigma(\sigma_l^{(3)}(z^l))\Sigma(\delta^{l+1})J(z^l)^2 + \Sigma(\sigma_l''(z^l))\Sigma(\delta^{l+1})J(z^l)\right).
\end{aligned} \tag{3.34}$$

The feedforward and backpropagation algorithms (2.5), (3.14), (3.20), (3.23), (3.27), and (3.32) can be used to solve any second-order PDE, both linear and non-linear. To compute mixed derivatives, (3.27) and (3.32) have to be modified by taking the appropriate partial derivative of (3.14), and (3.23). Higher-order PDEs can be solved by repeated differentiation of the feedforward and backpropagation algorithms. Note that to compute second-order derivatives, the weighted inputs, partial derivatives of the weighted inputs, activations, and partial derivatives of the activations needs to be re-computed or stored from the previous forward and backward passes.

4 Numerical examples

In this section we provide some concrete examples concerning how to apply the modified feedforward and backpropagation algorithms to model problems. Each problem requires their own modified backpropagation algorithms depending on the differential operator L. We start by some simple examples in some detail and later show some examples of more complicated problems.

In all the following numerical examples we have implemented the gradients according to the schemes presented in the previous section, and we consequently use the BFGS [7] method, with default settings, from `SciPy` [17] to train the ANN. We have tried all gradient free and gradient based numerical optimization methods available in the `scipy.optimize` package, and the online, batch, and stochastic gradient descent methods. BFGS shows superior performance compared to all other methods we have tried.

An issue one might encounter is that the line search in BFGS fails due to the Hessian matrix being very ill-conditioned. To circumvent this, we use a BFGS in combination with stochastic gradient descent. When BFGS fails due to line search failure, we run 1000 iterations with stochastic gradient descent with a very small learning rate, in the order of 10^{-9}, to bring BFGS out of the troublesome region. This procedure is repeated until convergence, or until the maximum number of allowed iterations has been exceeded.

4.1 Linear advection in 1D

The stationary, scalar, linear advection equation in 1D is given by

$$\begin{aligned} Lu = \frac{du}{dx} &= f, \quad 0 < x \leq 1, \\ u(0) &= g_0. \end{aligned} \tag{4.1}$$

To get an analytic solution we take, for example,

$$u = \sin(2\pi x)\cos(4\pi x) + 1, \tag{4.2}$$

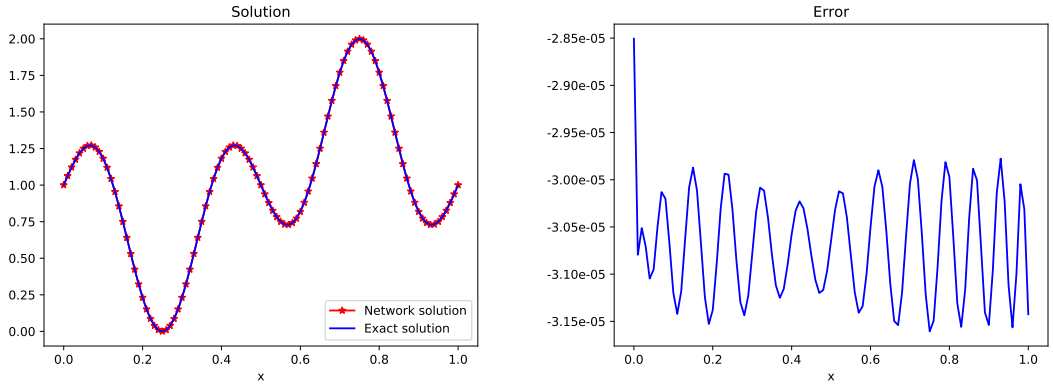
and plug it into (4.1) to compute f and g_0. In this simple case, the boundary data extension can be taken as the constant $G(x) \equiv u(0) = 1$ and the smoothed distance function can be seen in Figure 2a. In this simple case, we could take the distance function to be the line $D(x) = x$. When L acts on the ansatz $\hat{u}$ we get

$$L\hat{u} = \frac{dG}{dx} + \frac{dD}{dx}y_1^L + D\frac{\partial y_1^L}{\partial x}. \tag{4.3}$$

To use a gradient based optimization method we need the gradients of the quadratic residual cost function (3.2). They can be computed from (3.3) as

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^l} &= (L\hat{u} - f)\left(\frac{dD}{dx}\frac{\partial y_1^L}{\partial w_{jk}^l} + D\frac{\partial^2 y_1^l}{\partial x \partial w_{jk}^l}\right), \\ \frac{\partial C}{\partial b_j^l} &= (L\hat{u} - f)\left(\frac{dD}{dx}\frac{\partial y_1^L}{\partial b_j^l} + D\frac{\partial^2 y_1^l}{\partial x \partial b_j^l}\right). \end{aligned} \tag{4.4}$$

Each of the terms in (4.3) and (4.4) can be computed using the algorithms (2.5), (2.8), (3.14), (3.20), or (3.23). The result is shown in Figure 3. In this case, we used an ANN with 2 hidden layers, each with 10 neurons, and 100 collocation points.



(a) Exact and approximate ANN solution to the 1D stationary advection equation.

(b) The difference between the exact and approximated solution.

Figure 3: Approximate solution and error. The solution is computed using 100 equidistant points, using an ANN with 2 hidden layers with 10 neurons each.

4.2 Linear diffusion in 1D

The stationary, scalar, linear diffusion equation in 1D is given by

$$\begin{aligned} Lu = \frac{d^2u}{dx^2} = f, \quad & 0 < x < 1, \\ u(0) = g_0, \qquad & u(1) = g_1. \end{aligned} \tag{4.5}$$

To get an analytical solution we let, for example,

$$u = \sin(\frac{\pi x}{2}) \cos(2\pi x) + 1, \tag{4.6}$$

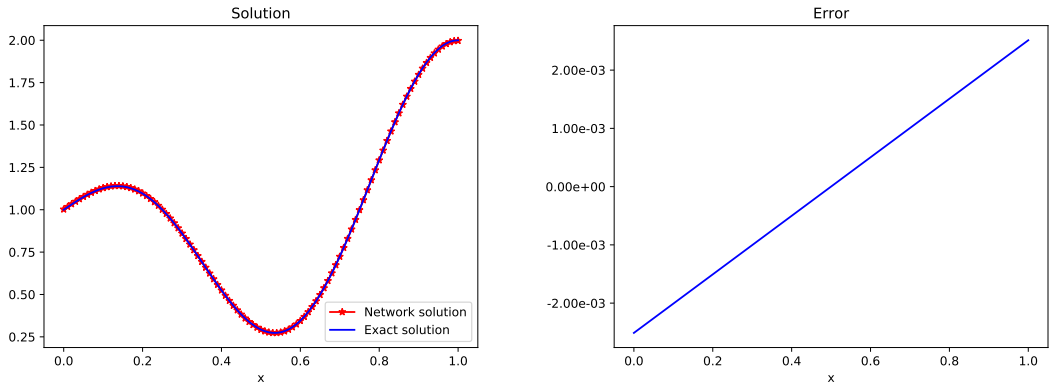
and compute f, g_0, and g_1. In this simple case, the boundary data extension can be taken as the straight line $G(x) = x + 1$. The smoothed distance function D is seen in Figure 2b. Here, we could instead take the smoothed distance function to be the parabola $D(x) = x(1 - x)$. When L acts on the ansatz we get

$$L\hat{u} = \frac{d^2G}{dx^2} + \frac{d^2D}{dx^2} y_1^L + 2\frac{dD}{dx}\frac{\partial y_1^L}{\partial x} + D\frac{\partial^2 y_1^L}{\partial x^2}, \tag{4.7}$$

and to use gradient based optimization we compute (3.3) as

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^l} &= (L\hat{u} - f)\left(\frac{d^2D}{dx^2}\frac{\partial y_1^L}{\partial w_{jk}^l} + 2\frac{dD}{dx}\frac{\partial^2 y_1^L}{\partial x \partial w_{jk}^l} + D\frac{\partial^3 y_1^L}{\partial x^2 \partial w_{jk}^l}\right), \\ \frac{\partial C}{\partial b_j^l} &= (L\hat{u} - f)\left(\frac{d^2D}{dx^2}\frac{\partial y_1^L}{\partial b_j^l} + 2\frac{dD}{dx}\frac{\partial^2 y_1^L}{\partial x \partial b_j^l} + D\frac{\partial^3 y_1^L}{\partial x^2 \partial b_j^l}\right). \end{aligned} \tag{4.8}$$

Each of the terms in (4.7) and (4.8) can be computed using the algorithms (2.5), (3.14), (3.20), (3.23), (3.27), and (3.32). The solution is shown in Figure 4, again using an ANN with two hidden layers with 10 neurons each, and 100 collocation points.



(a) Exact and approximate ANN solution to the 1D stationary diffusion equation.

(b) The difference between the exact and approximated solution.

Figure 4: Approximate solution and error. The solution is computed using 100 equidistant points, using an ANN with two hidden layers with 10 neurons each.

4.3 A remark on 1D problems

First- and second order problems in 1D have a rather interesting property — the boundary data can be added in a pure post processing step. A first order problem in 1D requires only one boundary condition, and the boundary data extension can be taken as a constant. When the first-order operator acts on the ansatz, the boundary data extension vanishes and is not used in any subsequent training. A second order problem in 1D requires boundary data at both boundary points, and the boundary data extension can be taken as the line passing through these points. When the second-order operator acts on the ansatz, the boundary data extension vanishes and is not used in any subsequent training. For one dimensional problems we hence let the ansatz be given by

$$\tilde{u}(x) = D(x)y^L(x), \tag{4.9}$$

and we can then evaluate the solution for any boundary data using

$$\hat{u}(x) = G(x) + \tilde{u}(x), \tag{4.10}$$

without any re-training. Even 1D equations can be time consuming for complicated problems and large networks, and being able to experiment with different boundary data without any re-training is an improvement.

4.4 Linear advection in 2D

The stationary, scalar, linear advection equation in 2D is given by

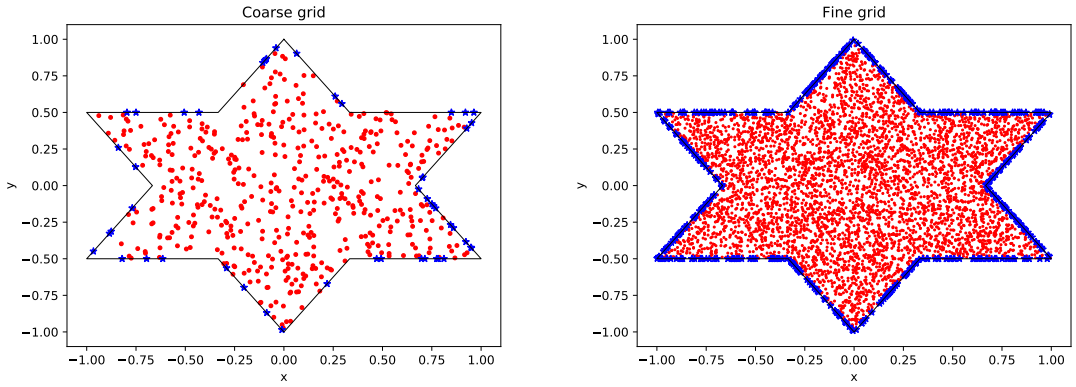
$$\begin{aligned} Lu = a\frac{\partial u}{\partial x} + b\frac{\partial u}{\partial y} &= f, \quad x \in \Omega, \\ u &= g, \qquad\quad x \in \Gamma, \end{aligned} \tag{4.11}$$

where a, b are the (constant) advection coefficients. The set $\Gamma \subset \partial\Omega$ is the part of the boundary where boundary conditions should be imposed. Let $n = n(x)$ denote the outer unit normal to $\partial\Omega$ at $x \in \partial\Omega$. Following [19], we have

$$\Gamma = \{x \in \partial\Omega : (a, b) \cdot n(x) < 0\}. \tag{4.12}$$

The condition (4.12), usually called the inflow condition, is easily incorporated into the computation of the distance function. Whenever we generate a boundary point to use in the computation of the distance function, we check if (4.12) holds, and if it does not, we add the point to the set of collocation points.

As an example we take the domain Ω to be a star shape and generate N uniformly distributed points inside Ω, and M uniformly distributed points on $\partial\Omega$, see Figure 5. Once the ANNs have been trained, we can evaluate on any number of points.



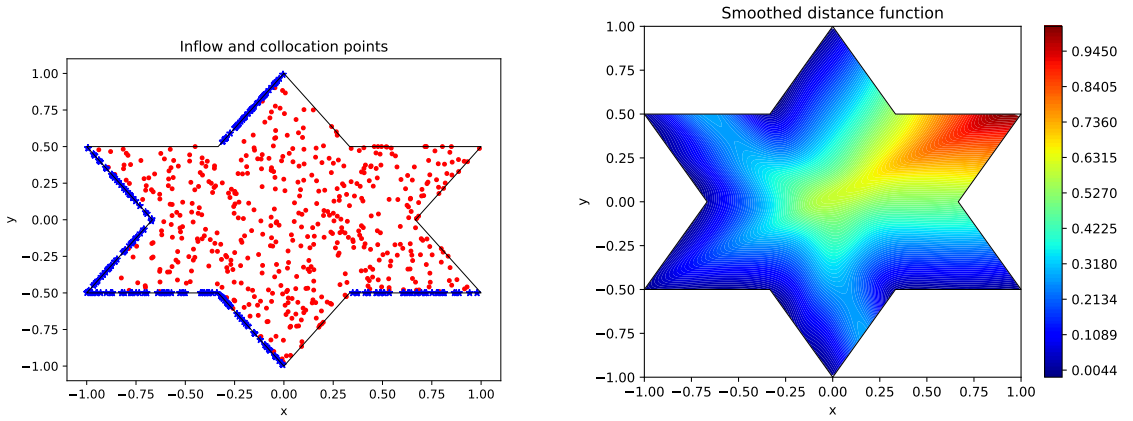
(a) $N = 500$ and $M = 50$ for computation of the smoothed distance function.

(b) $N = 5000$ and $M = 500$ for computation of the boundary data extension and solution.

Figure 5: Star-shaped domains with uniformly distributed points. The coarse grid is a fixed subset of the fine grid.

To compute the distance function, we compute (3.9) on the finer grid shown in Figure 5b, after traversing the boundary and moving all points that does not satisfy

(4.12) to the set of collocation points. To compute the smoothed distance function we train an ANN on the coarser grid shown in Figure 5a, and evaluate it on the finer grid. The results can be seen in Figure 6.



(a) Collocations and boundary points satisfying the inflow condition used to compute $d(x)$.

(b) Smoothed distance function, $D(x)$, computed on the coarse grid, using a single hidden layer with 20 neurons.

Figure 6: Inflow points and smoothed distance function for the advection equation in 2D, with advection coefficients $a = 1$ and $b = 1/2$.

When the advection operator L acts on the ansatz we get

$$L\hat{u} = a\left(\frac{\partial G}{\partial x} + \frac{\partial D}{\partial x}y_1^L + D\frac{\partial y_1^L}{\partial x}\right) + b\left(\frac{\partial G}{\partial y} + \frac{\partial D}{\partial y}y_1^L + D\frac{\partial y_1^L}{\partial y}\right). \tag{4.13}$$

To compute the solution we need the gradients of the residual cost function, given here by

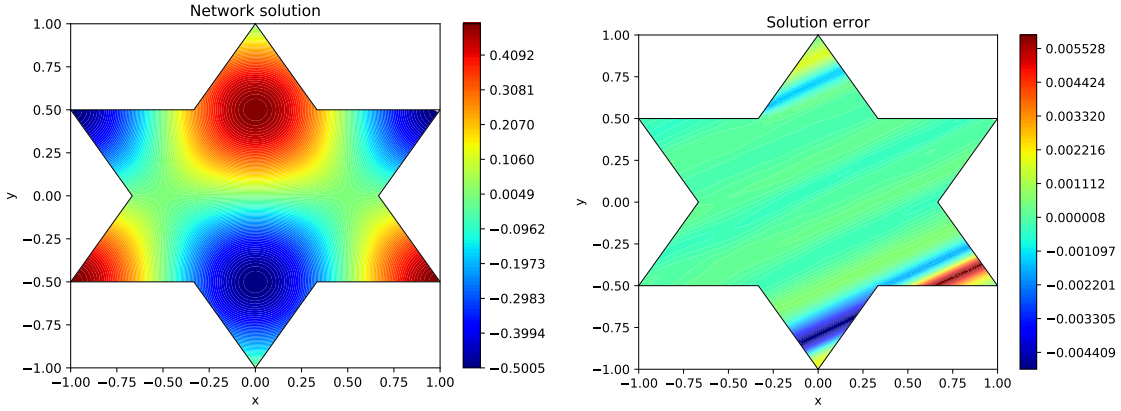
$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}^l} &= (L\hat{u} - f)\left(a\left(\frac{\partial D}{\partial x}\frac{\partial y_1^L}{\partial w_{ij}^l} + D\frac{\partial^2 y_1^L}{\partial x \partial w_{ij}^l}\right) + b\left(\frac{\partial D}{\partial y}\frac{\partial y_1^L}{\partial w_{ij}^l} + D\frac{\partial^2 y_1^L}{\partial y \partial w_{ij}^l}\right)\right), \\
\frac{\partial C}{\partial b_j^l} &= (L\hat{u} - f)\left(a\left(\frac{\partial D}{\partial x}\frac{\partial y_1^L}{\partial b_j^l} + D\frac{\partial^2 y_1^L}{\partial x \partial b_j^l}\right) + b\left(\frac{\partial D}{\partial y}\frac{\partial y_1^L}{\partial b_j^l} + D\frac{\partial^2 y_1^L}{\partial y \partial b_j^l}\right)\right).
\end{aligned} \tag{4.14}$$

Each of the terms in (4.13) and (4.14) can be computed using the algorithms (2.5), (2.8), (3.14), (3.20), or (3.23) as before. Note that the full gradient vectors can be computed simultaneously, and there is no need to do a separate feedforward and backpropagation pass for each component of the gradient.

To get an analytic solution we let, for example,

$$u = \frac{1}{2}\cos(\pi x)\sin(\pi y), \tag{4.15}$$

and plug it into (4.11) to compute f and g. An ANN with a single hidden layer with 20 neurons is used to compute the extension of the boundary data, G. To compute the solution we use an ANN with five hidden layers, each with 10 neurons. The result can be seen in Figure 7. Note that the errors follow the so-called streamlines. This is a well-known problem in FEM where streamline artificial diffusion is added to reduce the streamline errors [16].



(a) ANN solution to the 2D stationary advection equation.

(b) Difference between the exact and computed solution.

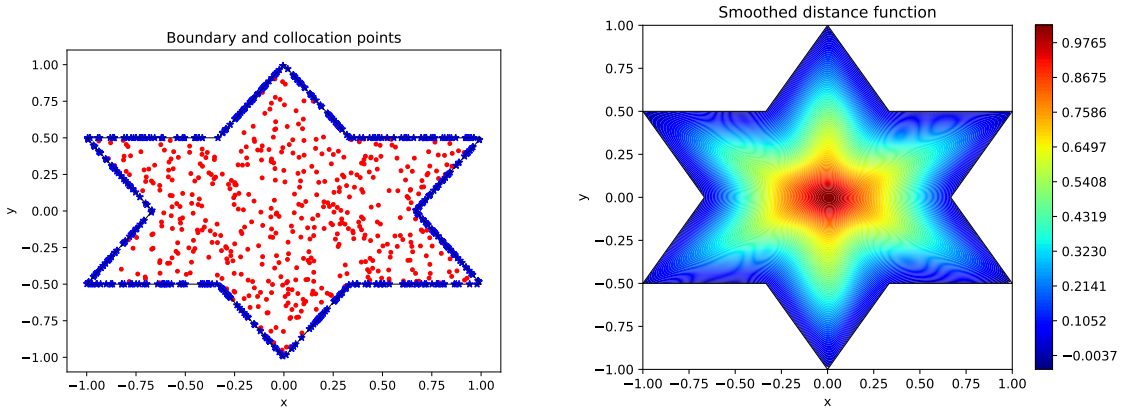
Figure 7: Solution and error for the advection equation in 2D, with advection coefficients $a = 1$ and $b = 1/2$, using five hidden layers with 10 neurons each.

4.5 Linear diffusion in 2D

The linear, scalar diffusion equation in 2D is given by

$$\begin{aligned} Lu = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= f, \quad x \in \Omega, \\ u &= g, \qquad\quad x \in \Gamma. \end{aligned} \tag{4.16}$$

In this case, $\Gamma = \partial\Omega$ and the smoothed distance function is computed using all points along the boundary as seen in Figure 8.



(a) Collocation and boundary points used to compute $d(x)$.

(b) Smoothed distance function, $D(x)$, computed on the coarse grid, using a single hidden layer with 20 neurons.

Figure 8: Boundary points and the smoothed distance function for the 2D diffusion equation.

To get an analytic solution to compare with, we let, for example,

$$u = \exp(-(2x^2 + 4y^2)) + \frac{1}{2}, \tag{4.17}$$

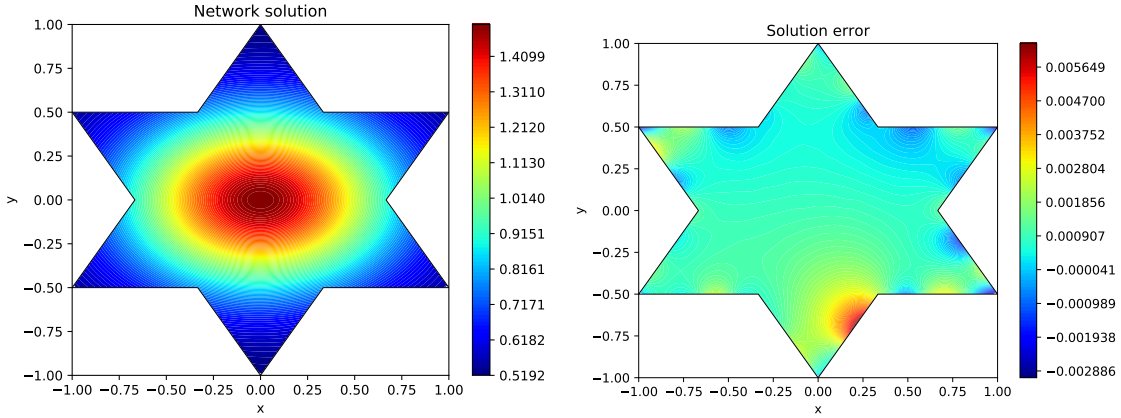
and use it to compute f and g in (4.16). The extension of the boundary data is again computed using an ANN with a single hidden layer with 20 neurons, and trained on all boundary points of the fine mesh. When the diffusion operator acts on the ansatz we get

$$\begin{aligned} L\hat{u} = &\frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 D}{\partial x^2} y_1^L + 2\frac{\partial D}{\partial x}\frac{\partial y_1^L}{\partial x} + D\frac{\partial^2 y_1^L}{\partial x^2} \\ &+ \frac{\partial^2 G}{\partial y^2} + \frac{\partial^2 D}{\partial y^2} y_1^L + 2\frac{\partial D}{\partial y}\frac{\partial y_1^L}{\partial y} + D\frac{\partial^2 y_1^L}{\partial y^2}, \end{aligned} \tag{4.18}$$

and the gradients of the residual cost function becomes

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}^l} &= (L\hat{u} - f)\left(\frac{\partial^2 D}{\partial x^2}\frac{\partial y_1^L}{\partial w_{jk}^l} + 2\frac{\partial D}{\partial x}\frac{\partial^2 y_1^L}{\partial x \partial w_{jk}^l} + D\frac{\partial^3 y_1^L}{\partial x^2 \partial w_{jk}^l}\right) \\ &\quad + (L\hat{u} - f)\left(\frac{\partial^2 D}{\partial y^2}\frac{\partial y_1^L}{\partial w_{jk}^l} + 2\frac{\partial D}{\partial y}\frac{\partial^2 y_1^L}{\partial y \partial w_{jk}^l} + D\frac{\partial^3 y_1^L}{\partial y^2 \partial w_{jk}^l}\right), \\ \frac{\partial C}{\partial b_j^l} &= (L\hat{u} - f)\left(\frac{\partial^2 D}{\partial x^2}\frac{\partial y_1^L}{\partial b_j^l} + 2\frac{\partial D}{\partial x}\frac{\partial^2 y_1^L}{\partial x \partial b_j^l} + D\frac{\partial^3 y_1^L}{\partial x^2 \partial b_j^l}\right) \\ &\quad + (L\hat{u} - f)\left(\frac{\partial^2 D}{\partial y^2}\frac{\partial y_1^L}{\partial b_j^l} + 2\frac{\partial D}{\partial y}\frac{\partial^2 y_1^L}{\partial y \partial b_j^l} + D\frac{\partial^3 y_1^L}{\partial y^2 \partial b_j^l}\right).\end{aligned} \tag{4.19}$$

Each of the terms in (4.19) and (4.18) can be computed using the algorithms (2.5), (3.14), (3.20), (3.23), (3.27), and (3.32). The solution, using an ANN with five hidden layers with 10 neurons each, can be seen in Figure 9.



(a) ANN solution to the 2D stationary diffusion equation.

(b) Difference between the exact and computed solution.

Figure 9: Solution and error for the diffusion equation in 2D, using five hidden layers with 10 neurons each.

4.6 Higher-dimensional problems

Higher-dimensional problems follow the same pattern as the 2D case. For example, in N dimensions we have the diffusion operator acting on the ansatz as

$$L\hat{u} = \sum_{i=1}^{N} \left(\frac{\partial^2 G}{\partial x_i^2} + \frac{\partial^2 D}{\partial x_i^2} y_1^L + 2 \frac{\partial D}{\partial x_i} \frac{\partial y_1^L}{\partial x_i} + D \frac{\partial y_1^L}{\partial x_i^2} \right), \tag{4.20}$$

and the gradients of the residual cost function becomes

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^l} &= (L\hat{u} - f) \sum_{i=1}^{N} \left(\frac{\partial^2 D}{\partial x_i^2} \frac{\partial y_1^L}{\partial w_{jk}^l} + 2 \frac{\partial D}{\partial x_i} \frac{\partial^2 y_1^L}{\partial x_i \partial w_{jk}^l} + D \frac{\partial^3 y_1^L}{\partial x_i^2 \partial w_{jk}^l} \right), \\ \frac{\partial C}{\partial b_j^l} &= (L\hat{u} - f) \sum_{i=1}^{N} \left(\frac{\partial^2 D}{\partial x_i^2} \frac{\partial y_1^L}{\partial b_j^l} + 2 \frac{\partial D}{\partial x_i} \frac{\partial^2 y_1^L}{\partial x_i \partial b_j^l} + D \frac{\partial^3 y_1^L}{\partial x_i^2 \partial b_j^l} \right). \end{aligned} \tag{4.21}$$

As we can see, the computational complexity increases linearly with the number of space dimensions as there is one additional term added for each dimension. The number of parameters in the deep ANN increases with the number of neurons in the first hidden layer for each dimension. For a sufficiently deep ANN, this increase is negligible.

The main cause for increased computational time is the number of collocation points. Regular grids grows exponentially with the number of dimensions and quickly becomes infeasible. Uniformly random numbers in high-dimensions are in fact not uniformly distributed across the whole space, but tends to cluster on hyperplanes, thus providing a poor coverage of the whole space. This is a well-known fact for Monto Carlo methods, and led to the development of sequences of quasi-random numbers called low discrepancy sequences. See for example ch. 2 in [32], or [4, 35] and references therein.

There are plenty of different low discrepancy sequences, and an evaluation of their efficiency in the context of deep ANNs is beyond the scope of this paper. A common choice is the Sobol sequence [34] which is described in detail in [2] for $N \leq 40$, and later extended in [14] and [15] for $N \leq 1111$, and $N \leq 21201$, respectively. To compute a high-dimensional problem we thus generate N_d points from a Sobol sequence in N dimensions to use as collocation points, and N_b points in $N-1$ dimensions to use as boundary points. The rest follows the same procedure as previously described.

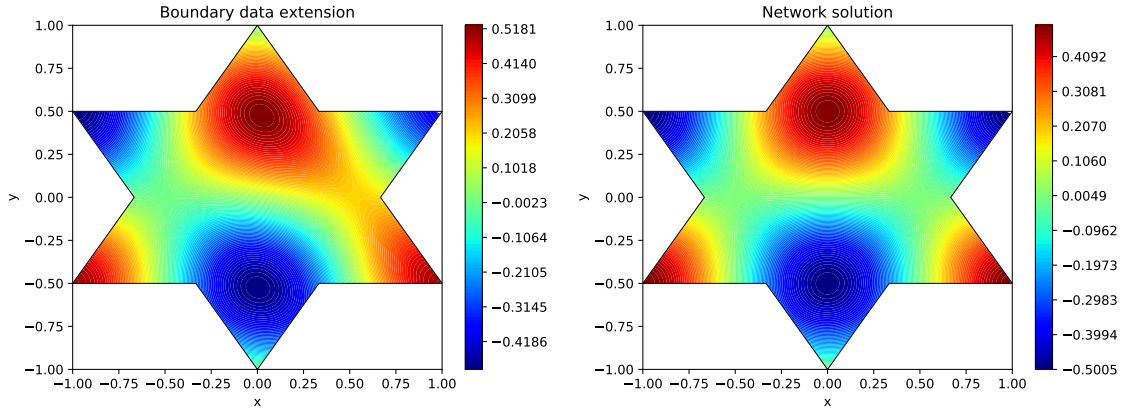
A few more examples of work on high-dimensional problems and deep neural networks for PDEs can be found in [33, 10, 6].

5 Convergence considerations

Training neural networks consumes a lot of time and computational resources. It is therefore desirable to reduce the number of required iterations until a certain accuracy has been reached. In the following we will discuss two factors which strongly influence the number of required iterations. The first is pre-training the network using the available boundary data, and the second is to increase the number of hidden layers.

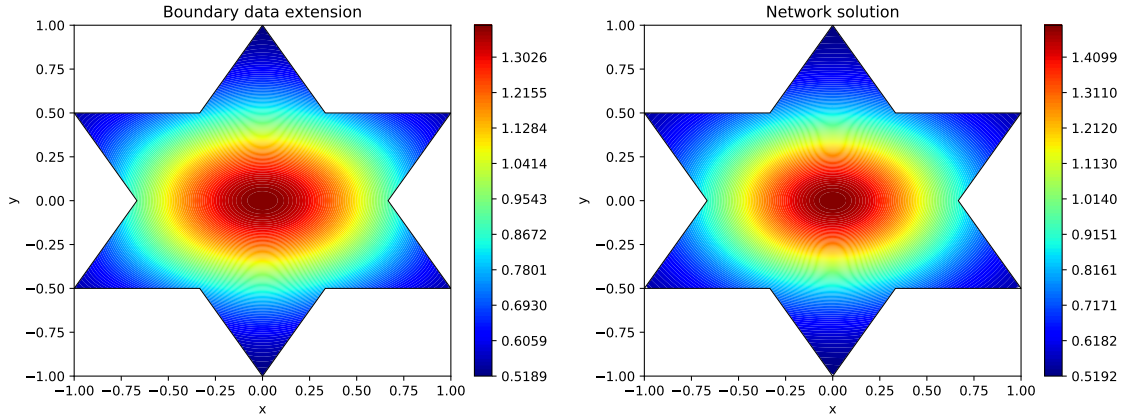
5.1 Boundary data pre-training

As previously described, we used a low-capacity ANN to compute the global extension of the boundary data to save some computational time. However, the low-capacity ANN that extends the boundary data is already an approximate solution to the PDE. Figure 10 shows the boundary and solution ANNs evaluated on all collocation and boundary points. Note that the boundary ANN has not been trained on a single collocation point inside the domain.



(a) Boundary data extension for the advection equation.

(b) Network solution of the advection equation.



(c) Boundary data extension for the diffusion equation.

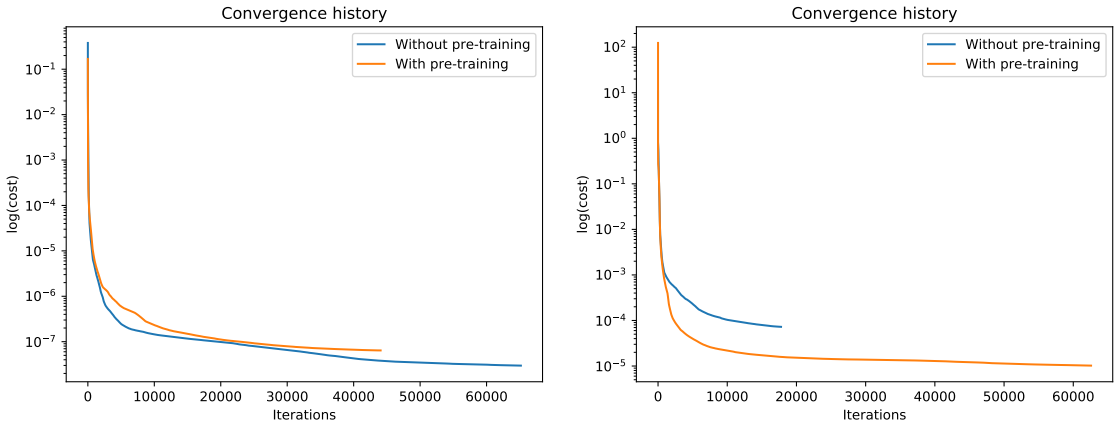
(d) Network solution of the diffusion equation.

Figure 10: The boundary ANNs use a single hidden layer with 20 neurons, trained only on the boundary points. The solution ANNs use five hidden layers with 10 neurons each, trained on both the boundary and collocation points.

As fitting an ANN to the boundary data is an order of magnitude faster than solving the PDE, this suggests that we can pre-train the solution ANN by fitting it to the boundary data. It is still a good idea to keep the boundary ANN as a separate low-capacity ANN as it will speed up the training of the solution ANN due to the many evaluations that is required during training.

When training with the BFGS method, pre-training has limited effect due to the efficiency of the method to quickly reduce the value of the cost function. For less

efficient methods, such as gradient descent, the difference can be quite pronounced as boundary data pre-training gives a good starting points for the iterations. There is, however, one prominent effect. The time until the first line search fail is greatly reduced by boundary data pre-training for the diffusion problem which is significantly more ill-conditioned than the advection problem. In Figure 11 we show the convergence history of BFGS until the first line search fail, with and without pre-training. Line search fails is the main bottleneck of the BFGS method, and reducing them is a major gain.



(a) Convergence history for the advection equation.

(b) Convergence history for the diffusion equation.

Figure 11: Convergence history of BFGS until the first line search fail, with and without boundary data pre-training.

5.2 Number of hidden layers

Most earlier works, as for example in the pioneering work by Lagaris et al. [22, 23], have been focused on neural networks with a single hidden layer. Single hidden layer networks are sufficient to approximate any continuous function to any accuracy by having a sufficient amount of neurons in the hidden layer [12, 13, 24]. However, by fixing the capacity of the network and increasing the number of hidden layers, we can see a dramatic decrease in the number of iterations required to reach a desired level of accuracy.

In the following, we solve the diffusion problem in 2D as before (with boundary data pre-training) using networks with 1, 2, 3, 4 and 5 hidden layers with 120, 20, 14, 12, and 10, neurons in each hidden layer, respectively. This gives networks with 481,

501, 477, 517, and 481 trainable parameters (weights and biases), respectively, which we consider comparable. The networks are trained until the cost is less than 10^{-5}, and we record the number of required iterations. The result can be seen in Figure 12. We can clearly see how the number of required iterations decreases as we increase the number of hidden layers. At five hidden layers, we are starting to experience the vanishing gradient problem [8], and the convergence starts to deteriorate. Note that using a single hidden layer we did not reach the required tolerance until we exceeded the maximum number of allowed iterations.

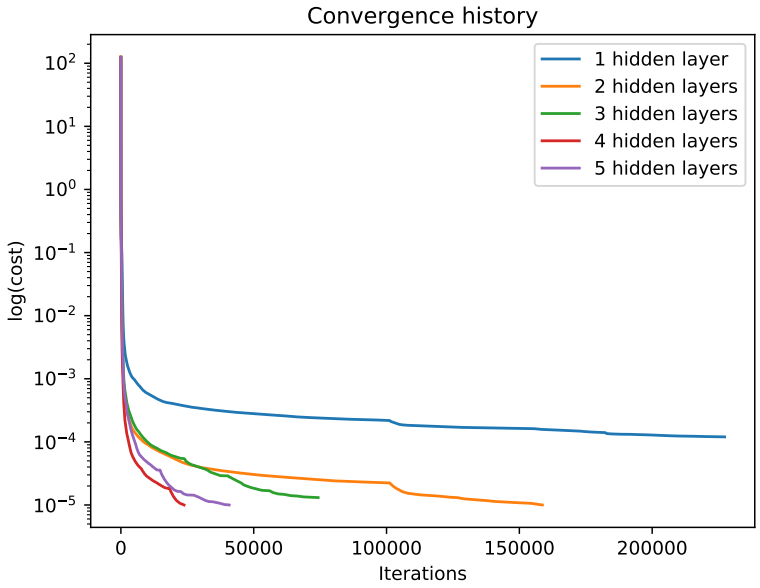


Figure 12: Number of iterations required for increasing network depth.

6 Summary, conclusion and future research

We have presented a method for solving advection and diffusion type partial differential equations in complex geometries using deep feedforward artificial neural networks. We have derived analytical expressions for the gradients of the cost function with respect to the network parameters, and the gradients of the network itself with respect to the input, for arbitrarily deep networks. By following our derivations, one can extend this work to include non-linear systems of PDEs of arbitrary order with mixed derivatives.

We conducted numerical experiments with different network depths and it is clear that increasing the number of hidden layers is beneficial for the number of training iterations required to reach a certain accuracy. The effects will be even more pronounced when adding deep learning techniques that prevents the vanishing gradient problem. Extending the feedforward network to more complex configurations, using for example convolutional layers, dropout, and batch normalization is the topic of future work to develop even deeper networks and study the layer-wise variations when changing different parts of the PDE.

As a more speculative note on future research we recall that is a well-know fact, in for example image analysis, that the different layers of deep ANN are responsible for detecting different features [9, 27, 28]. One can pose the question if something similar occurs in the context of PDEs. In general a boundary value problem for a PDE consists of the following features: a differential operator (the PDE), geometry, a boundary operator and boundary data. An interesting problem could be to study the effect on the different layers when varying the features of the PDE.

For example, consider the 2D diffusion problem,

$$\begin{aligned} k\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) &= f, \quad x \in \Omega(s), \\ u &= \tau g, \quad x \in \partial\Omega(s), \end{aligned} \tag{6.1}$$

where we have parameterized the differential operator by a diffusion coefficient k, the domain and the boundary data by s, $\tau \in [0,1]$, respectively. The shape parameter s transforms the domain from one shape to another. We can then compute the variance of the weights and biases as we are varying k, s, τ by,

$$\begin{aligned} \mathrm{Var}(W^l) &= \sum_i \frac{||W_i^l - W_{i-1}^l||_F}{||W_0^l||_F}, \\ \mathrm{Var}(b^l) &= \sum_i \frac{||b_i^l - b_{i-1}^l||}{||b_0^l||}, \end{aligned} \tag{6.2}$$

where, for example, $||\cdot||_F$ is the Frobenius norm, and $||\cdot||$ the usual Euclidean vector norm, and W_i^l, b_i^l denotes the weights and biases of layer l for $k = k_i$, $s = s_i$, or $\tau = \tau_i$.

Experiments show that the variances defined are very sensitive to the local minimum found by the optimizer and we have not been able to clearly see the layer-wise variance. Moreover, in the case of feedforward ANN the vanishing gradient problem causes the layers to train at different speeds which makes it difficult to separate a

poor local minimum from the actual effect of varying the different features. However, it could still possible to train a base network and to re-train only some layers depending on which feature of the PDE that is changing. Our results suggest that one should strive to train even deeper networks by deploying a variety of methods and techniques that have been developed over the past decade to overcome the difficulties in training deep networks.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[2] P. Bratley and B. L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14(1):88–100, Mar. 1988.

[3] P. Chaudhari, A. Oberman, S. Osher, S. Soatto, and G. Carlier. Deep relaxation: partial differential equations for optimizing deep neural networks. *ArXiv e-prints*, Apr. 2017.

[4] J. Cheng and M. J. Druzdzel. Computational investigation of low-discrepancy sequences in simulation algorithms for Bayesian networks. *ArXiv e-prints*, Jan. 2013.

[5] N. E. Cotter. The Stone–Weierstrass theorem and its application to neural networks. *IEEE Transactions on Neural Networks*, 1(4):290–295, Dec. 1990.

[6] W. E, J. Han, and A. Jentzen. Deep learning-based numerical methods for high–dimensional parabolic partial differential equations and backward stochastic differential equations. *ArXiv e-prints*, June 2017.

[7] R. Fletcher. *Practical methods of optimization*. Wiley, 2nd edition, 1987.

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

[9] I. J. Goodfellow, Q. V. Le, A. M. Saxe, H. Lee, and A. Y. Ng. Measuring invariances in deep networks. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, pages 646–654. Curran Associates Inc., 2009.

[10] J. Han, A. Jentzen, and W. E. Overcoming the curse of dimensionality: Solving high-dimensional partial differential equations using deep learning. *ArXiv e-prints*, July 2017.

[11] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

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

[13] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551–560, 1990.

[14] S. Joe and F. Y. Kuo. Remark on algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 29(1):49–57, Mar. 2003.

[15] S. Joe and F. Y. Kuo. Constructing Sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing*, 30(5):2635–2654, 2008.

[16] C. Johnson and J. Saranen. Streamline diffusion methods for the incompressible Euler and Navier–Stokes equations. *Mathematics of Computation*, 47(175):1–18, 1986.

[17] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org`, 2001–. Accessed 2017-07-07.

[18] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. *ArXiv e-prints*, June 2017.

[19] H.-O. Kreiss. Initial boundary value problems for hyperbolic systems. *Communications on Pure and Applied Mathematics*, 23(3):277–298, 1970.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[21] M. Kumar and N. Yadav. Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: A survey. *Computers & Mathematics with Applications*, 62(10):3796–3811, 2011.

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

[23] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou. Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, 11(5):1041–1049, Sept. 2000.

[24] X. Li. Simultaneous approximations of multivariate functions and their derivatives by neural networks with one hidden layer. *Neurocomputing*, 12(4):327–343, 1996.

[25] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec. 1943.

[26] K. S. McFall and J. R. Mahan. Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions. *IEEE Transactions on Neural Networks*, 20(8):1221–1233, Aug. 2009.

[27] G. Montavon, M. L. Braun, and K.-R. Müller. Kernel analysis of deep networks. *J. Mach. Learn. Res.*, 12:2563–2581, Nov. 2011.

[28] K. R. Müller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201, Mar. 2001.

[29] S. M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute, 1989.

[30] K. Rudd and S. Ferrari. A constrained integration (cint) approach to solving partial differential equations using artificial neural networks. *Neurocomputing*, 155:277–285, 2015.

[31] K. Rudd, G. Muro, and S. Ferrari. A constrained backpropagation approach for the adaptive solution of partial differential equations. *IEEE Transactions on Neural Networks and Learning Systems*, 25(3):571–584, 2014.

[32] R. Seydel. *Tools for Computational Finance*. Universitext (1979). Springer, 2004.

[33] J. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *ArXiv e-prints*, Aug. 2017.

[34] I. M. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[35] X. Wang and I. H. Sloan. Low discrepancy sequences in high dimensions: How well are their projections distributed? *J. Comput. Appl. Math.*, 213:366–386, Mar. 2008.

[36] N. Yadav, A. Yadav, and M. Kumar. *An Introduction to Neural Network Methods for Differential Equations*. Springer Briefs in Applied Sciences and Technology. Springer Netherlands, 2015.

View publication stats