# NEURAL NETWORK-BASED SOLVERS FOR PDES

MARIA CAMERON

## Contents

## 1. What is a neural network?

A *feed-forward fully connected neural network* with $l$ *hidden layers* is a composition of functions of the form

$$\mathcal{N}(x;\theta) = \mathcal{L}_{l+1}\circ\sigma_l\circ\mathcal{L}_l\circ\sigma_{l-1}\circ\ldots\circ\sigma_1\circ\mathcal{L}_1. \tag{1}$$

The symbol $\mathcal{L}_k$ denotes the $k$'s affine operator of the form $\mathcal{L}_k(x) = A_kx + b_k$, while $\sigma_k$ denotes a nonlinear function called an *activation function*. The activation functions act on vector arguments entry-wise.

The activation functions as well as the dimensions of matrices $A_k$ and the number of hidden layers $l$ are chosen by the user and called the *hyperparameters* of the neural network.

The matrices $A_k$ and shift vectors (or bias vectors) $b_k$ are encoded into the argument $\theta$: $\theta = \{A_k, b_k\}_{k=1}^{l+1}$. They are called the *parameters* of the neural network. The term *training neural network* means optimizing $\{A_k, b_k\}_{k=1}^{l+1}$ with respect to a certain objective function called the *loss function*. The loss function is set up by the user so that it is minimized if the neural network $\mathcal{N}(x;\theta)$ satisfies certain conditions. For example, one might want the neural network to assume certain values $f_j$ at certain points $x_j$, $j = 1,\ldots,N$. These points $x$ are called the *training data*. In this case, a common choice of the loss function is the least squares error:

$$\mathsf{Loss}(x;\theta) = \frac{1}{N}\sum_{j=1}^{n}\|\mathcal{N}(x_j;\theta) - f_j\|^2. \tag{2}$$

The activation functions $\sigma_k$ can be arbitrary **non-polynomial** functions. Popular choices are (Fig. 1)

- sigmoid:

$$\sigma(x) = \frac{1}{1+e^{-x}};$$

- hyperbolic tangent:

$$\tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}; \tag{3}$$

- rectified linear unit:

$$\mathsf{ReLU}(x) = \begin{cases} x, & x \ge 0 \\ 0, & x < 0. \end{cases}$$

There are many other possible choices for the activation functions.

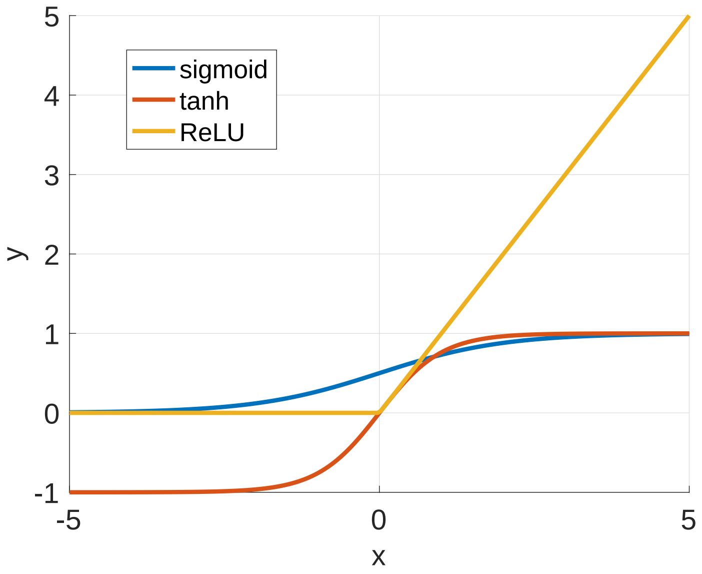

FIGURE 1. Common activation functions for neural networks.

There are many other types of neural networks – see e.g. the textbook "Deep Learning" by I. Goodfellow, Y. Bengio, and A. Courville.

## 2. What is special about neural networks?

Neural networks with at least one hidden layer can approximate any continuous function on a compact set provided that the activation function satisfies some nonrestrictive conditions – see below. Moreover, they can approximate it together with its derivatives. The approximation theory for neural networks has been a subject of active research starting from the 1980s. A beautiful paper proving very important results and giving a comprehensive account of the work in this field is the one by Allan Pinkus (1999) available via the UMD library. The key contributors to the development of the approximation theory for neural networks in the late 1980s are

- Carrol and Dickinson,
- Cybenko,
- Hornik, Stinchcombe, and White,
- Funahashi.

The main result established by Hornik et al. (1989) is the following.

**Theorem 1.** *The set*

$$M(\sigma) := \mathsf{span}\{\sigma(a\cdot x + b),\ a \in \mathbb{R}^d,\ b \in \mathbb{R}\} \tag{4}$$

*and* $\sigma : \mathbb{R} \to [0,1]$ *is a non-decreasing function such that* $\lim_{y\to-\infty}\sigma(y) = 0$*,* $\lim_{y\to\infty}\sigma(y) = 1$*, is dense in the set of continuous functions* $C(\mathbb{R}^d)$ *on any compact set.*

Note that the activation function $\sigma$ does not need to be continuous. A suitable choice of $\sigma$ is the Heaviside function $h(x) = 1$ if $x \geq 0$ and $h(x) = 0$ if $x < 0$.

The following theorems are proven in Pinkus (1999):

**Theorem 2.** *Let* $\sigma$ *be any continuous function on* $\mathbb{R}$*. Then*

$$M(\sigma) := \mathsf{span}\{\sigma(a\cdot x + b),\ a \in \mathbb{R}^d,\ b \in \mathbb{R}\}$$

*is dense in* $C(\mathbb{R}^d)$ *in the topology of uniform convergence on compacta, if and only if* $\sigma$ *is not a polynomial.*

It is easy to see why $\sigma$ should not be a polynomial. Indeed, if $\sigma$ is a polynomial of degree $r$ then $M(\sigma)$ will be the set of all polynomials of degree $\leq r$ which is not dense in $C(\mathbb{R}^d)$.

Moreover, in the same work, Pinkus proved an approximation result with derivatives which is a very important theoretical basis for looking for the solutions to PDEs in the form of neural networks.

**Theorem 3.** *If* $\sigma : \mathbb{R} \to \mathbb{R}$ *is* $m$ *times continuously differentiable and* $\sigma$ *is not a polynomial then*

$$M(\sigma) := \mathsf{span}\{\sigma(a\cdot x + b),\ a \in \mathbb{R}^d,\ b \in \mathbb{R}\}$$

*is dense in* $C^{m^1,\ldots,m^s}(\mathbb{R}^d)$*, i.e., for any* $f \in C^{m^1,\ldots,m^s}(\mathbb{R}^d)$*, any compact set* $K \subset \mathbb{R}^d$*, and any* $\epsilon > 0$*, there is* $g \in M(\sigma)$ *such that*

$$\max_{x\in K} |D^k f(x) - D^k g(x)| < \epsilon$$

*for all* $k \in \mathbb{Z}^n_+$ *for which* $k \leq m^i$ *for some* $i$*.*

This means that neural networks with just one hidden layer can approximate any continuous function together with its derivatives on a compact set provided that the activation function is not a polynomial and is smooth enough, and the neural network is wide enough.

This does not mean that this approximation is efficient, i.e., that high accuracy can be achieved with a few terms in the linear combination. Pinkus shows that it is advantageous to use neural networks with two hidden layers because they can efficiently approximate functions with compact supports.

## 3. Connection to approximations by trigonometric functions in 1D

It is well-known (see e.g. W. Strauss, ”Partial Differential Equations”) that any twice continuously differentiable function $f$ on $[r_1, r_2]$ satisfying $f(r_1) = f(r_2) = 0$ can be approximated by a Fourier sine series

$$f(x) \approx \sum_{k=1}^{N} \alpha_k \sin\left(\frac{\pi k(x - r_1)}{r_2 - r_1}\right) \quad \text{where} \quad \alpha_k = \frac{2}{L}\int_{r_1}^{r_2} f(x) \sin\left(\frac{\pi k(x - r_1)}{r_2 - r_1}\right) dx. \tag{5}$$

Moreover, the Fourier series converges to $f$ uniformly on $[r_1, r_2]$ as $N \to \infty$, i.e.,

$$\lim_{N\to\infty} \max_{x\in[r_1,r_2]} \left| f(x) - \sum_{k=1}^{N} \alpha_k \sin\left(\frac{\pi k(x - r_1)}{r_2 - r_1}\right)\right| = 0. \tag{6}$$

Let us reconcile (5) and $\mathsf{span}\{\sigma(a \cdot x + b),\ a \in \mathbb{R},\ b \in \mathbb{R}\}$. Obviously,

$$\sum_{k=1}^{N} \alpha_k \sin\left(\frac{\pi k(x - r_1)}{r_2 - r_1}\right) \in \mathsf{span}\left\{\sin\left(\frac{\pi k x}{r_2 - r_1} - \frac{r_1}{r_2 - r_1}\right),\ k \in \mathbb{N}\right\},$$

i.e.,

$$\sigma \equiv \sin, \quad a \in \left\{\frac{\pi k}{r_2 - r_1},\ k \in \mathbb{N}\right\} \quad \text{and} \quad b = -\frac{r_1}{r_2 - r_1} \text{ is fixed.}$$

Hence,

$$\mathsf{span}\left\{\sin\left(\frac{\pi k x}{r_2 - r_1} - \frac{r_1}{r_2 - r_1}\right),\ k \in \mathbb{N}\right\} \subset \mathsf{span}\{\sin(a \cdot x + b),\ a \in \mathbb{R},\ b \in \mathbb{R}\}.$$

This shows that the approximation of $f$ in $\mathsf{span}\{\sin(a \cdot x + b),\ a \in \mathbb{R},\ b \in \mathbb{R}\}$ cannot be less accurate than the approximation of $f$ by the sine Fourier series.

## 4. Connection to finite elements in 2D

Reference: Juncai He, Lin Li, Jinchao Xu, *ReLU Deep Neural Networks from the Hierarchical Basis Perspective, Computer & Mathematics with Applications, Vol. 120, 15 August 2022, pp. 105–114, arXiv:2105.04156*.

Let us fix a triangulation with the set of nodes $(x_j, y_j)$, $1 \le j \le N$, and fineness $h$ in a rectangular domain $\Omega \subset \mathbb{R}^2$ constructed out of a regular rectangular mesh as shown in Fig. 2. We define the set of finite element basis functions

$$\eta_j(x, y) = \begin{cases} 1, & (x, y) = (x_j, y_j), \\ 0, & (x, y) = (x_i, y_i),\ i \neq j, \\ \text{linear}, & \text{on each triangle} \\ \text{continuous}, & (x, y) \in \Omega. \end{cases} \tag{7}$$

It follows from the classical approximation theory that any twice continuously differentiable function $f : \Omega \to \mathbb{R}$ can be approximated by its linear interpolant $I_h f$ with the set of

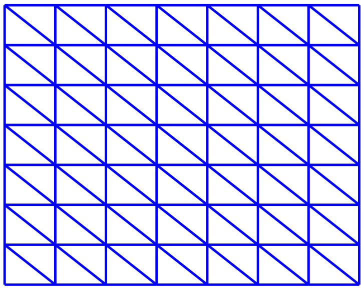

FIGURE 2. The hat-function $g$ and its composition with itself $g_2$.

collocation points $(x_j, y_j)$, $1 \le j \le N$, with accuracy $O(h^2)$. Now we will follow Juncai He, Lin Li, Jinchao Xu (2021) to show that $f$ can be approximated by a 2-layer ReLU neural network with $15N$ neurons per layer.

The hat function $g : \mathbb{R} \to \mathbb{R}$ is defined by

$$g(x) = \begin{cases} 2x, & x \in [0, 1/2), \\ 2(1-x), & x \in [1/2, 1], \\ 0, & \text{otherwise}. \end{cases} \tag{8}$$

The function $g_2 : \mathbb{R} \to \mathbb{R}$ is defined as

$$g_2(x) = g(g(x)). \tag{9}$$

The graphs of $g$ and $g_2$ are shown in Fig. 3.

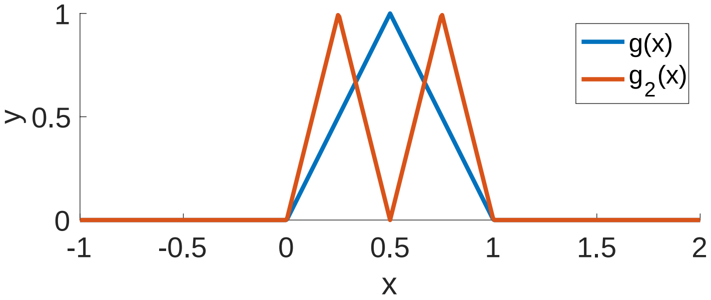

FIGURE 3. The hat-function $g$ and its composition with itself $g_2$.

It is easy to check that $g(x)$ can be expressed as a linear combination of three ReLU functions:

$$g(x) = 2\mathsf{ReLU}(x) - 4\mathsf{ReLU}(x - 1/2) + 2\mathsf{ReLU}(x-1). \tag{10}$$

Indeed,

$$2\mathsf{ReLU}(x) - 4\mathsf{ReLU}(x - 1/2) + 2\mathsf{ReLU}(x-1) = \begin{cases} 0, & x \le 0, \\ 2x, & x \in [0, 1/2), \\ 2x - 4x + 2 = 2(1-x), & x \in [1/2, 1], \\ 2x - 4x + 2 + 2x - 2 = 0, & x > 1 \end{cases} = g(x).$$

Therefore, the function $g_2$ can be represented as a two-layer ReLU neural network with three neurons in each layer. In fact, $g_2$ can be represented as a linear combination of five ReLU functions, i.e., as a one-layer ReLU neural network with 5 neurons:

$$g_2(x) = 4\mathsf{ReLU}(x) - 8\mathsf{ReLU}(x - 1/4) + 8\mathsf{ReLU}(x - 1/2) - 8\mathsf{ReLU}(x - 3/4) + 4\mathsf{ReLU}(x-1). \tag{11}$$

Now we consider the standard FEM basis function $\phi(x, y)$ on the unit square $[0, 1]^2$ shown in Fig. 4. The basis functions $\eta_j(x, y)$ (7) are obtained from $\phi(x, y)$ via appropriate affine transformations:

$$\eta_j(x, y) = \phi\left(\frac{x - x_j}{h}, \frac{y - y_j}{h}\right). \tag{12}$$

One can verify that $\phi(x, y)$ can be expressed as a linear combination of three $g_2$ functions:

$$\phi(x, y) = \frac{1}{2}\left[g_2\left(\frac{x}{2}\right) + g_2\left(\frac{y}{2}\right) - g_2\left(\frac{x+y}{2}\right)\right]. \tag{13}$$

The problem with representation (13) is that it is valid on $[0, 1]^2$ but not globally. The right-hand side in (13) is equal to $1/2$ at $(3/2, 3/2)$ rather than 0. The global representation of $\phi(x, y)$ can be constructed by applying the function

$$\mathsf{ReLU1}(x) = \mathsf{ReLU}(x) - \mathsf{ReLU}(x-1) = \begin{cases} 0, & x < 0, \\ x, & x \in [0, 1], \\ 1, & x > 1, \end{cases} \tag{14}$$

to the arguments in the right-hand side of (13):

$$\phi(x, y) = \frac{1}{2}\left[g_2\left(\frac{\mathsf{ReLU1}(x)}{2}\right) + g_2\left(\frac{\mathsf{ReLU1}(y)}{2}\right) - g_2\left(\frac{\mathsf{ReLU1}(x) + \mathsf{ReLU1}(y)}{2}\right)\right]. \tag{15}$$

Eqs. (11) and (15) imply that the linear interpolant $I_h f(x, y)$ of a function $f$

$$I_h f(x, y) = \sum_{j=1}^{N} f(x_j, y_j)\eta_j(x, y) = \sum_{j=1}^{N} f(x_j, y_j)\phi(T_j(x, y)) \tag{16}$$

can be represented as a two-layer neural network with at most $15N$ neurons per layer.

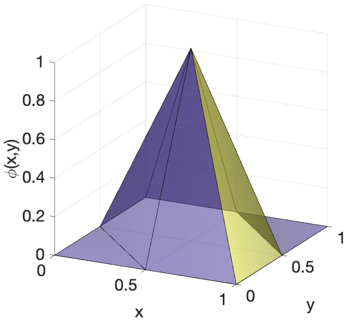

Figure 4. The standard basis finite element hat-function in 2D.

## 5. Remarks on recent advances

The recent surge of interest in neural networks in the numerical PDE community is caused by several factors, in particular by the fact that neural networks allow us to approximate functions defined in high-dimensional domains. In this connection, the question of great interest is whether there is the *curse of dimensionality*. More precisely, how does the number of parameters of neural networks representing functions defined in high-dimensional domains scale with the dimension of the domain? It is well-known that the number of mesh points required for representing a function in a $d$-dimensional domain scales as $O(h^{-d})$ where $h$ is the mesh step, i.e., exponentially.

Significant contributions to the approximation theory concerning this question are recently done by Haizhao Yang and collaborators. In Deep Network Approximation for Smooth Functions, Lu et al. (2021), it is shown that the number of required parameters for approximating a function $f \in C^s([0,1]^d)$ with a neural network with ReLU activation function and $L$ hidden layers of width $N$ scales as

$$O\left(\|f\|_{C^s([0,1]^d)} N^{-2s/d} L^{-2s/d}\right).$$

This formula shows that the number of required parameters grows exponentially as $d \to \infty$. On the other hand, in Neural network approximation: Three hidden layers are enough by Shen et al. 2021, its is shown that if one takes advanced specially crafted activation functions $\sigma_1$, $\sigma_2$, and $\sigma_3$ and constructs a neural network with three hidden layers of width $N$, then the number of required parameters scales as $\sqrt{d}$ with the dimension $d$.

## 6. Application to solving PDEs

The idea of solving PDEs with the aid of neural networks is the following. We propose a solution model for a PDE that involves a neural network. We choose an appropriate loss function so that if the loss function is zero then the PDE is satisfied in a given set of training points. Then we train the neural network.

This approach to solving ODEs and PDEs dates back to the work "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations" by I. Lagaris, A. Likas, and D. Fotiadis (1998). At that time, this approach was far from the mainstream. A boom of the development of neural network-based PDE solvers started from works by Weinan E and collaborators in the late 2010s. The key contributors are:

- Weinan E and collaborators,
- Jianfeng Lu, Lexing Ying, Yehaw Khoo,
- George Karniadakis, Paris Perdikars, Maziar Raissi.

This boom is facilitated by the availability of packages for training neural networks that encode the automatic differentiation the most popular of which are PyTorch and TensorFlow.

Neural network-based solvers have several advantages.

- Numerical solutions to PDEs found in the form of neural networks with smooth activation functions are *globally defined smooth functions* unlike finite difference and finite element solutions.
- Numerical solutions to PDEs can be easily differentiated.
- The neural network framework allows to construct loss functions that pick the desired solutions in the case of multiple solutions.
- Neural networks allow us to approximate functions defined in high-dimensional domains. Hence neural network-based PDE solvers are amenable to promotion to high dimensions.

### 6.1. An application to solving PDEs using NNs.

We consider Problem 5 from "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations" by I. Lagaris, A. Likas, and D. Fotiadis (1998). Consider the following BVP for the Poisson equation:

$$u_{xx} + u_{yy} = \phi(x,y), \quad (x,y) \in \Omega = [0,1]^2, \tag{17}$$

$$u(0,y) = f_0(y), \quad u(1,y) = f_1(y), \quad u(x,0) = g_0(x), \quad u(x,1) = g_1(x). \tag{18}$$

Lagaris, Likas, and Fotiadis (1998) proposed to look for the solution to (17)–(18) in the following form:

$$\Psi(x,y,\mathsf{w}) = A(x,y) + h(x)h(y)\mathcal{N}(x,y,\mathsf{w}). \tag{19}$$

Here $A(x,y)$ is a function satisfying boundary conditions (18) which can be written out analytically:

$$\begin{aligned} A(x,y) = (1-x)f_0(y) + xf_1(y) + (1-y)\left[g_0(x) - \{(1-x)f_0(0) + xf_1(0)\}\right] \\ + y\left[g_1(x) - \{(1-x)f_0(1) + xf_1(1)\}\right]. \end{aligned} \tag{20}$$

The function $h$ is chosen to guarantee that the second term in the right-hand side in (19) is zero on the boundary $\partial\Omega$:

$$h(t) = t(1-t). \tag{21}$$

The function $\mathcal{N}(x, y, \mathsf{w})$ is a neural network (NN) with one hidden layer and a single linear output unit:

$$\mathcal{N}(x, y, \mathsf{w}) = \mathbf{v}^{\top}\sigma(W\mathbf{x} + \mathbf{u}), \quad \mathsf{w} \equiv (\mathbf{v}, W, \mathbf{u}), \tag{22}$$

where

$$\mathbf{x} \equiv \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{v} \in \mathbb{R}^N, \quad W = (w_{ij}) \in \mathbb{R}^{N\times 2}, \quad \mathbf{u} \in \mathbb{R}^N,$$

and $\sigma$ is a nonlinear function applied entry-wise. We will experiment with

$$\sigma(t) = (1 + \exp(-t))^{-1} \text{ (sigmoid) and } \sigma(t) = \tanh(t).$$

I chose this example due to its simplicity. A one-layer NN is sufficient to achieve quite impressive accuracy using just a few training points, and the derivatives of $\mathcal{N}$ with respect to $x$, $y$, and parameters packed in $\mathsf{w}$ can be computed analytically without the use of *automatic differentiation*.

As the form of the solution is set, the proposed parameter-dependent solution is plugged into the differential operator, evaluated at a number of training points, and equated to the corresponding values of the right-hand side of the differential equation. Then the nonlinear least-squares problem (NLLS) is set up to minimize the sum of the squares of the discrepancies by choosing an optimal set of the parameters. The nonlinear least-squares problem for the PDE above is

$$f(\mathsf{w}) \coloneqq \frac{1}{2}\sum_{j=1}^{n} |\Psi_{xx}(x_j, y_j, \mathsf{w}) + \Psi_{yy}(x_j, y_j, \mathsf{w}) - f(x_j, y_j)|^2 \to \min. \tag{23}$$

Plugging $\Psi$ into PDE (17) we get:

$$\begin{aligned} r_j(\mathsf{w}) &= \Psi_{xx}(x_j, y_j, \mathsf{w}) + \Psi_{yy}(x_j, y_j, \mathsf{w}) - f(x_j, y_j) \\ &= A_{xx} + A_{yy} + \left[h(x)h''(y) + h''(x)h(y)\right]\mathcal{N} \\ &+ 2\left[h'(x)h(y)\mathcal{N}_x + h(x)h'(y)\mathcal{N}_y\right] \\ &+ h(x)h(y)\left[\mathcal{N}_{xx} + \mathcal{N}_{yy}\right] - f(x_j, y_j). \end{aligned} \tag{24}$$

Equation (24) shows that in order to solve the NLLS (23), one needs to calculate the first derivatives of $\mathcal{N}$, $\mathcal{N}_x$, $\mathcal{N}_y$, $\mathcal{N}_{xx}$, and $\mathcal{N}_{yy}$ with respect to the components of $\mathbf{v}$, $W$, and $\mathbf{u}$.

For the convenience of programming, we denote the parameters $u_j$ by $w_{3j}$, $W_{jk}$, $k = 0, 1$, by $w_{0j}$ and $w_{1j}$, and $v_j$ by $w_{2j}$ and write the neural network with one hidden layer and $N = 10$ neurons of the form:

$$\mathcal{N}(x, y; w) = \sum_{j=0}^{N-1} w_{3j}\sigma\left(w_{0j}x + w_{1j}y + w_{2j}\right), \tag{25}$$

where $\sigma(z)$ is a nonlinear activation function acting entrywise. We will use $\sigma(z) = \tanh(z)$ *hyperbolic tangent*. The total number of parameters to optimize is $4n$.

It is easy to check that

$$\mathcal{N}_x(x, y; w) = \sum_{j=0}^{N-1} w_{3j}w_{0j}\sigma'\left(w_{0j}x + w_{1j}y + w_{2j}\right),$$

$$\mathcal{N}_y(x,y;w) = \sum_{j=0}^{N-1} w_{3j}w_{1j}\sigma'\left(w_{0j}x + w_{1j}y + w_{2j}\right),$$

$$\mathcal{N}_{xx}(x,y;w) = \sum_{j=0}^{N-1} w_{3j}w_{0j}^2\sigma''\left(w_{0j}x + w_{1j}y + w_{2j}\right),$$

$$\mathcal{N}_{yy}(x,y;w) = \sum_{j=0}^{N-1} w_{3j}w_{1j}^2\sigma''\left(w_{0j}x + w_{1j}y + w_{2j}\right).$$

The derivatives of $\mathcal{N}$, $\mathcal{N}_x$, $\mathcal{N}_y$, $\mathcal{N}_{xx}$, and $\mathcal{N}_{yy}$ with respect to the components of $w$ are easily found from these formulas.

We set the right-hand side and the boundary functions in (17)–(18) to:

$$\phi(x,y) = e^{-x}(x - 2 + y^3 + 6y),$$

$$f_0(y) = y^3, \qquad f_1(y) = (1 + y^3)e^{-1},$$

$$g_0(x) = xe^{-x}, \qquad g_1(x) = e^{-1}(x+1).$$

Then the exact solution is

$$u(x,y) = e^{-x}(x + y^3).$$

The training set is a $5 \times 5$ set of mesh points (see Fig. 5). The number of hidden units $N$ is set to 10. This makes the dimensionality of the parameter space equal to 40. The function $\sigma = \tanh$:

```
fun = @(x)tanh(x);
dfun = @(x)1./cosh(x).^2;
d2fun = @(x)-2*sinh(x)./cosh(x).^3;
d3fun = @(x)(4*sinh(x).^2-2)./cosh(x).^4;
```

The initial guess for parameter values was the vector of all ones. We solve the resulting nonlinear least squares problem using the Levenberg-Marquardt (LM) algorithm. This is a powerful deterministic optimizer of trust-region type for nonlinear least squares problems. The test points are the $101 \times 101$ mesh points. Stopping criteria were: either the number of iterations exceeds 120, or the norm of the gradient of the objective function decays to $10^{-4}$.

```
LM:
iter # 109: f = 0.00000002701552, |df| = 4.4645e-05
CPUtime = 1.260592e-01
max|err| = 1.301953e-06
L2 err = 4.449413e-05
```

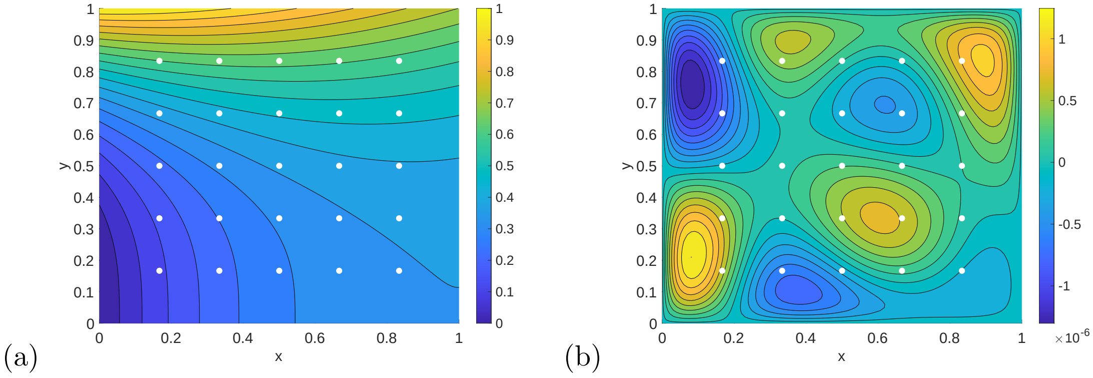

Figure 5. The solution (a) to the NLLS (23) and the error (b) committed by Levenberg-Marquardt. While dots are the training points.