FYS-STK4155 H23 Project 3:
Solving Poisson's PDE using neural networks

Davor Dundovic

26.1.2023

Abstract

This project implements two distinct methods for solving Poisson's equation $-\nabla u = f$ with Dirichlet boundary conditions. The first method employs finite differences to discretise the problem and utilises Jacobi and Gauss-Seidel schemes to iteratively converge to the sought solution. Analytical studies of convergence rates guide optimal convergence. The second, data-driven method, seeks a solution that satisfies boundary conditions by construction and involves a neural network to learn the underlying physical principle for the domain interior. The method is implemented in the Tensorflow machine learning library and is carefully validated. We observe fast convergence for functions u that exhibit a more gradual variation away from the boundaries, such as for $u_1 = e^{-x}(x + y^3)$, compared to more challenges with oscillatory functions such as $u_2 = \sin(2\pi x)\sin(3\pi y)$. Finally, the project explores varied neural network configurations, revealing the efficacy of $x + \sin^2(x)$ activation functions for learning periodic functions and the superiority of single hidden layer models for capturing the periodicity of u_2.

1 Introduction

In the field of Applied Mathematics, the study of partial differential equations (PDEs) stands as a cornerstone for understanding the dynamic evolution of physical phenomena across science and engineering. PDEs serve as fundamental tools for describing complex systems such as fluid dynamics, heat transfer, electromagnetic fields, and more. However, closed-form solutions for PDEs are a rarity, typically often achieved under idealised conditions that rarely capture the full complexity of real-world scenarios. The scarcity of analytical solutions becomes particularly pronounced when dealing with nonlinear, multi-dimensional, or coupled PDEs. In the absence of closed-form solutions, scientists and engineers turn to numerical methods to approximate solutions, such as finite difference or finite element methods. These methods then lead to a large sparse system of linear equations that must be solved efficiently on a computer, which often presents a computational bottleneck in simulations. Indeed, developing efficient algorithms for solving large linear systems is a very active fields of research, with multigrid method being one of the most efficient methods at the moment (Briggs, Henson, and McCormick 2000). However, the rise of data science techniques offers exciting prospects to aid and improve existing traditional numerical methods (Greenfeld et al. 2019; Pereira et al. 2020) or to even completely replace them, as we demonstrate here.

Poisson's equation is the canonical elliptic PDE traditionally used to model the distribution of electrostatic potential in a given region of space, the potential given a mass distribution in the Newtonian gravitation theory, and the temperature in a heat conduction problem. Solutions to the Poisson equation can exhibit a wide array of dynamic behaviours, depending on the specific nature of the source term. More recent advancements have seen the Poisson equation emerge in innovative applications, such as image processing (Pérez, Gangnet, and Blake 2003, 2023), which continues to add to the equation's significance, as well as to finding effective techniques to solving it. Traditional numerical methods to solve the Poisson problem, such as Jacobi and Gauss-Seidel iterative methods, are integral to the history of solving PDEs like the Poisson equation, and are the basis of state-of-the-art multigrid method (Briggs, Henson, and McCormick 2000). While not the primary focus of this report, the motivation to explore them here is their simplicity, easy implementation, and rigorous mathematical theory that backs them. Due to their wide acceptance and basic theoretical underpinning, such methods are often used as a benchmark for comparison with more advanced numerical practices.

In this report we adopt one such advanced numerical practice: by solving the Poisson problem using (deep) neural networks. Such an approach is still unconventional and often inferior to traditional methods (Blechschmidt and Ernst 2021). Indeed, the scientific community is taking longer to adopt machine learning in their applications, especially considering how machine learning has transformed almost every aspect of human activity. Arguably, this lag in adopting machine learning is mostly due to the rigorous nature that scientific computing and prediction has when we have to consider the laws of physics that are at play. Machine learning methods still often lack the maturity to satisfy such rigorous scientific scrutiny (Blechschmidt and Ernst 2021). However, with the increasing demands in sciences, there is now a drive to adopt machine learning in traditional scientific communities to try and combine large data sets that are emerging and also to combine that with the improvements in High Performance Computing, which have enabled us to build gigantic models to understand the world around us.

Unlike in conventional machine learning problems, the mechanism, or the governing equations, behind modelling a physical system is usually known *a priori*. Indeed, we have a wealth of knowledge from decades of research in physics-based modelling which we should not discard while using data-driven approaches. There are several ways of incorporating this pre-existing knowledge into our machine learning model. An increasingly growing field with such goals is that of *Physics-Informed Machine Learning* (Karniadakis et al. 2021; Chris Rackauckas et al. 2019; Christopher Rackauckas et al. 2020). What is more, Karniadakis et al. (2021) controversially claim that neural networks offer considerable advantages in solving PDEs due to their inherent flexibility, which allows them to operate effectively in complex, uncertain, and high-dimensional problem spaces where traditional numerical methods falter.

2 Theory and implementation

2.1 Traditional iterative methods

2.1.1 Finite difference discretisation

For the purposes of demonstrating the basic iterative methods of Jacobi and Gauss-Seidel, we consider Poisson's equation on a unit square domain Ω with homogeneous Dirichlet conditions on the boundary $\partial\Omega$:

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega = (0,1)^2, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{1}$$

where $\nabla^2(\cdot)$ is the Laplace operator, $f = f(x, y)$ is the given real-valued source term, and $u = u(x, y)$ is the sought real-valued function.

In order to solve any PDE on a computer, we must first discretise it. In particular, we wish to write Poisson's eq. (1) as a linear system of finite equations with finite unknowns of the form

$$A\mathbf{U} = \mathbf{f}, \tag{2}$$

where A is the discretised Laplacian operator, $\mathbf{f}$ is the discretised source term, and $\mathbf{U}$ is the numerical approximation of the solution u. The focus of this case study is not in the discretisation methods, so throughout the report we use one of the simplest *finite difference* approximations. The finite difference method is very common throughout numerical analysis and is well represented in literature (e.g. LeVeque 2007; Iserles 2009; Trefethen and Bau 1997) so we do not discuss it in detail in this report.

We discretise our unit square domain with a uniform Cartesian grid of $(M+1)\times(N+1)$ grid points (x_i, y_j), where

$$\begin{aligned} x_i &= i\Delta x \qquad 0 \le i \le N, \\ y_j &= j\Delta y \qquad 0 \le j \le M, \end{aligned} \tag{3}$$

where $\Delta x = 1/N$, $\Delta y = 1/M$ are step sizes in x and y directions, respectively, and $N, M \ge 2$. Finally, we use the second-order centred finite difference approximation, commonly referred to as the *five-point formula*, for $\nabla^2 u = f$ at interior grid points (LeVeque 2007):

$$\frac{1}{\Delta x^2}(U_{i-1,j} - 2U_{i,j} + U_{i+1,j}) + \frac{1}{\Delta y^2}(U_{i,j-1} - 2U_{i,j} + U_{i,j+1}) = f_{i,j}, \tag{4}$$

where $i \in \{1, \dots, N-1\}$, $j \in \{1, \dots, M-1\}$, $U_{i,j}$ is the shorthand notation for $U(x_i, y_j)$ and similarly $f_{i,j} = f(x_i, y_j)$. Furthermore, the homogeneous Dirichlet boundary conditions translate into

$$\begin{aligned} U_{0,j} = 0,\ U_{N,j} = 0 \qquad 1 \le j \le M-1, \\ U_{i,0} = 0,\ U_{i,M} = 0 \qquad 0 \le i \le N. \end{aligned} \tag{5}$$

This is a system of $(N-1)(M-1) + 2(M-1) + 2(N+1) = (N+1)(M+1)$ equations for $(N+1)(M+1)$ unknowns. We can reduce this to a system of $(N-1)(M-1)$ equations by eliminating the four boundary terms given in eq. (5). By doing so, we can now replace the derivative in eq. (1) for interior grid points to give us a system of $(N-1)(M-1)$ equations and equally many unknowns. If we choose to write the right-hand side of $A\mathbf{U} = \mathbf{f}$ in a column-major order, i.e. with the columns stacked on top of each other from left to right so that $\mathbf{f} = (f_{1,1}, f_{2,1}, \dots, f_{M-1,1}, f_{M-1,2}, \dots, f_{M-1,N-1})^\top \in \mathbb{R}^{(N-1)(M-1)}$, then the coefficient matrix $A \in \mathbb{R}^{(N-1)(M-1)\times(N-1)(M-1)}$ is a large sparse block-diagonal matrix

$$A = \underbrace{\begin{pmatrix} B & C & & & \\ C & B & C & & \\ & \ddots & \ddots & \ddots & \\ & & C & B & C \\ & & & C & B \end{pmatrix}}_{N-1 \text{ matrices}} \left.\vphantom{\begin{pmatrix} B \\ C \\ \ddots \\ C \\ C \end{pmatrix}}\right\} N-1 \text{ matrices} \tag{6}$$

where empty elements represent null matrices, $B \in \mathbb{R}^{(M-1)\times(M-1)}$ is a tridiagonal matrix

$$B = \frac{1}{\Delta y^2}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} + 2C,$$

and $C = 1/\Delta x^2\, \mathbf{I}_{M-1}$ is a diagonal $(M-1)\times(M-1)$ matrix with $1/\Delta x^2$ on the diagonal (Demmel 1997).

The matrix A can be easily constructed with Kronecker products (Demmel 1997):

$$A = \frac{1}{\Delta y^2}\begin{pmatrix} 2 & -1 & \\ & \ddots & \\ & -1 & 2 \end{pmatrix} \otimes \mathbf{I}_{M-1} + \mathbf{I}_{N-1} \otimes \frac{1}{\Delta x^2}\begin{pmatrix} 2 & -1 & \\ & \ddots & \\ & -1 & 2 \end{pmatrix}. \tag{7}$$

Written in the form (7), it is easy to show that the eigenvalues of A are a sum of the eigenvalues of the two matrices written in full in (7), which is useful for examining spectral properties and convergence of traditional algorithms (Demmel 1997). However, this is outside of the scope of this project, and the reader is referred to appendix A and Demmel (1997) for details.

2.1.2 Jacobi and Gauss-Seidel iterative methods

As formulated in the previous subsection, we wish to solve the linear system $A\mathbf{U} = \mathbf{f}$ obtained from the five-point formula (4) for Poisson's equation (1). Specifically, we implement iterative methods of Jacobi and Gauss-Seidel, which can be described as a matrix splitting. Specifically, we split the matrix A as $A = E - F$, where E is nonsingular (Demmel 1997). This transforms the system $A\mathbf{U} = \mathbf{f}$ into (Demmel 1997)

$$\begin{aligned} E\mathbf{U} - F\mathbf{U} &= \mathbf{f} \\ \Rightarrow \quad \mathbf{U} &= \underbrace{E^{-1}F}_{C}\ \mathbf{U} + E^{-1}\mathbf{f}, \end{aligned}$$

where, after left-multiplying by E^{-1}, we obtained the iteration matrix $C = E^{-1}F$ of the method. Given an initial iterate $\mathbf{U}^{(0)}$, we can apply a fixed point iteration to get

$$\mathbf{U}^{(k+1)} = C\mathbf{U}^{(k)} + E^{-1}\mathbf{f} \tag{8}$$

as our iterative method. We are interested in the convergence of such methods, where we want the norm of the error

$$\mathbf{e}^{(k)} = \hat{\mathbf{U}} - \mathbf{U}^{(k)} \tag{9}$$

to converge to 0 as $k \to \infty$. Here $\hat{\mathbf{U}}$ is the *exact* discretised solution to Poisson's eq. (2) obtained by an in-built function, to which we wish to compare the solution $\mathbf{U}$ obtained by our own implementation. We use $\hat{\mathbf{U}}$ in the definition of the error in order to capture the inherent discretisation error, which is not captured by sampling the analytical solution u at the grid points.

Applying the iteration matrix to the error (9) gives us the relationship between error at each subsequent iteration: $\mathbf{e}^{(k+1)} = C\mathbf{e}^{(k)}$, which can be easily checked by direct substitution (Iserles 2009). It follows by induction that $\mathbf{e}^{(k)} = C^k\mathbf{e}^{(0)}$. It is easy to show (Theorem 6.1. in Demmel 1997) that the iteration (8) converges to the solution $\hat{\mathbf{U}}$ for any initial iterate $\mathbf{U}^{(0)}$ if and only if $\rho(C) < 1$, where $\rho(C) = \max_i |\lambda_i(C)|$ is the spectral radius of C. Furthermore, the rate of convergence of (8) is $r(C) \equiv -\log_{10}\rho(C)$ (Iserles 2009). Therefore, a good iterative method will be such that iteration (8) is easily computed and such that $\rho(C)$ is small.

The basic algorithm employed throughout this section is given in pseudo-code in algorithm 1. Worth noting is the calculation of the residual given in Line 7, where we are computing the infinity norm of the residual. Another option would be to look at the absolute difference of subsequent iterates $|\mathbf{U}^{(k)} - \mathbf{U}^{(k-1)}|$ and terminate the algorithm when the difference gets below set tolerance. Computing such difference costs $\mathcal{O}(MN)$ while computing the residual defined as the norm of (9) costs $\mathcal{O}((MN)^2)$, so the former may be preferred for large systems.

Algorithm 1 Basic Iterative Scheme

```
1: Construct $C$, $E^{-1}$
2: Choose initial iterate $\mathbf{u}_0$
3: Set maxiter, TOL
4: $k = 0$
5: while residual > TOL and k < maxiter do
6:     $\mathbf{U}^{(k+1)} = C\mathbf{U}^{(k)} + E^{-1}\mathbf{f}$
7:     residual $= \|A\mathbf{U}^{(k+1)} - \mathbf{f}\|_\infty$
8:     $k = k + 1$
```

In what follows, it will be useful for us to think of a splitting $A = D + L + R$, where D is the diagonal, L is the strict lower triangle and R is the strict upper triangle of A. Namely, choosing $E = D$ and therefore $F = -(L + R)$ yields the Jacobi method:

$$\mathbf{U}_{\mathrm{J}}^{(k+1)} = \underbrace{-D^{-1}(L+R)}_{C_J}\,\mathbf{U}_{\mathrm{J}}^{(k)} + D^{-1}\mathbf{f} \tag{10}$$

If we instead choose $E = D + L$ and $F = -R$ we get the Gauss-Seidel method

$$\mathbf{U}_{\mathrm{GS}}^{(k+1)} = -(D+L)^{-1}R\,\mathbf{U}_{\mathrm{GS}}^{(k)} + (D+L)^{-1}\mathbf{f}. \tag{11}$$

We refer the reader to Corollary 6.1. in Demmel (1997) for a proof that Jacobi and Gauss-Seidel methods are convergent and that, provided A is consistently ordered, $\rho(C_{GS}) = (\rho(C_J))^2$.

2.2 Lagaris method

Motivated by the course lecture notes (Hjorth-Jensen and Winther-Larsen, n.d.) that this report is a part of, we implement a method proposed by Lagaris, Likas, and Fotiadis (1998) to solve differential equations using artificial neural networks. Neural networks are a viable choice for such a task since any continuous function can be approximated to arbitrary accuracy by a neural network of at least one hidden layer with a finite number of parameters (Universal Approximation Theorem, see e.g. Hornik, Stinchcombe, and White 1989).

The Lagaris, Likas, and Fotiadis (1998) method is one of the earliest examples of physics-informed machine learning, as it incorporates existing physics knowledge into the construction of the machine learning model. The most common incorporation of this pre-existing domain knowledge into a machine learning model is to use the differential equation in the loss function of the neural network, which gives rise to a Physics-Informed Neural Network (PINN, see e.g. Raissi, Perdikaris, and Karniadakis 2019). This is the basis of the Lagaris, Likas, and Fotiadis (1998) method, which we outline below on the two-dimensional Poisson problem with Dirichlet boundary conditions. We refer the reader to the original paper for details and for demonstrations of the method on solving other differential equations and systems of differential equations.

Namely, we consider the Poisson problem in a unit square domain with Dirichlet boundary conditions:

$$\begin{aligned} -\nabla^2 u &= f \qquad && \text{in } \ \Omega = (0,1)^2 \\ u &= g_i && \text{on } \Gamma_i, \ i = 1,2,3,4, \end{aligned} \tag{12}$$

where $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ represent the different parts of the boundary $\partial\Omega$, and $g_i = g_i(x, y)$ are the corresponding boundary functions on each part Γ_i. Lagaris, Likas, and Fotiadis (1998) propose seeking a *trial solution*

$v = v(x, y; p)$ of the form

$$v = G + x(x-1)y(y-1)NN, \tag{13}$$

where $G = G(x, y)$[1] is a function manually chosen such that it satisfies the Dirichlet boundary conditions $v = g_i$ on Γ_i and $NN = NN(x, y; p)$ is the output of the neural network with trainable parameters p (weights and biases). Therefore, the trial function v satisfies $v = g_i$ on Γ_i boundary conditions by construction, since the term $x(x-1)y(y-1)$ vanishes on $\partial\Omega$. By substituing the trial solution v in eq. (12), the Poisson problem becomes:

$$\begin{aligned} -\nabla^2 v = f \quad & \text{in } \Omega = (0,1)^2 \\ v = G \quad & \text{on } \partial\Omega. \end{aligned} \tag{14}$$

By requiring the neural network to only learn the closure relation $-\nabla^2 v = f$, this approach may potentially result in faster training and more stable convergence. In contrast, PINNs that also have to learn boundary conditions require carefully chosen training data along the domain boundary to prevent potential issues related to the continuity and smoothness of the solution at the boundary (Christopher Rackauckas et al. 2020; Raissi, Perdikaris, and Karniadakis 2019; Lu et al. 2021).

2.2.1 Implementation and validation

While physics-informed neural networks for the solution of differential equations can be implemented in general machine learning libraries, such as Tensorflow (Martín Abadi et al. 2015) or PyTorch (Paszke et al. 2019), there has been a rise in purpose-specific libraries. These libraries offer an intuitive way for users to translate the mathematical description of the problem into highly-optimised code without requiring expertise with programming or machine learning. Such libraries simplify the formulation of loss functions, domain geometries, boundary conditions, and other necessary steps that are often tedious to implement in general-purpose libraries such as Tensorflow or PyTorch that were not originally designed with such applications in mind. For example, python package DeepXDE still relies on these libraries as its computational back-end for robust and highly-optimised tensor computations, while providing a user-friendly interface for defining physics-informed neural networks (Lu et al. 2021). On the other hand, NeuralPDE is part of the Julia-based SciML Software Ecosystem with their original implementation of the underlying methods (Zubov et al. 2021).

In this report, we chose to use Tensorflow to implement the ideas of Lagaris, Likas, and Fotiadis (1998). As a validation of our implementation, we begin by replicating original authors' results. Namely, we consider problem 5 from Lagaris, Likas, and Fotiadis (1998) where we solve the following Poisson's equation for $u = u(x, y)$:

$$-\nabla^2 u = -e^{-x}(-2 + x + 6y + y^3), \tag{15}$$

with boundary conditions $u(0, y) = y^3, u(1, y) = (1 + y^3)e^{-1}, u(x, 0) = xe^{-x}$, and $u(x, 1) = e^{-x}(x + 1)$. The analytic solution to the problem is given by $u = e^{-x}(x + y^3)$. We seek the trial solution v of the form (13), which requires an appropriate function G to satisfy the boundary conditions. However, the function G chosen in Lagaris, Likas, and Fotiadis (1998) does not satisfy the $y = 1$ boundary condition due to a sign error (see Appendix B). Therefore, here we correct the sign error which yields a function that correctly satisfies boundary conditions:

$$G = (1-x)y^3 + x(1+y^3)e^{-1} + (1-y)x(e^{-x} - e^{-1}) + y((1+x)e^{-x} - (1 - x + 2xe^{-1})). \tag{16}$$

Since the trial solution $v = G + x(x-1)y(y-1)NN$ satisfies boundary by construction, training the neural

1. Note that the original paper denotes the function G as A, which we here avoid to not confuse the reader with the matrix A introduced in section 2.1.

network only requires learning a closure relation $-\nabla^2 v = f$ by minimising the mean squared error loss

$$\mathcal{L} := \frac{1}{n}\sum_{i=1}^{n} |-\nabla^2 v(x_i, y_i) - f(x_i, y_i)|^2, \tag{17}$$

where $\{x_i, y_i\}_{i=1}^{n}$ denote n training points within the domain.

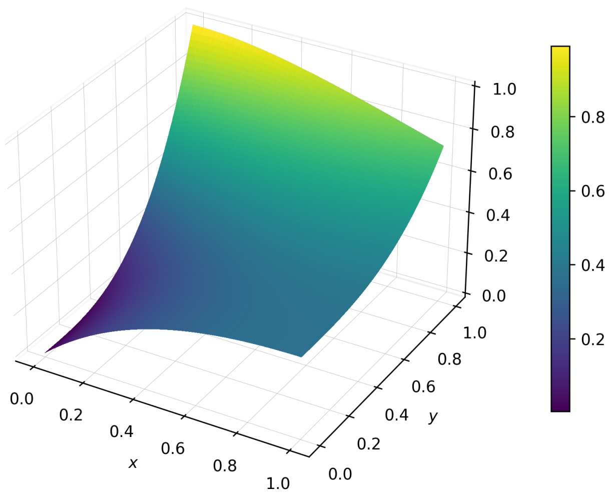
While Tensorflow provides numerous examples and a detailed documentation for building neural network models and modifying their architecture, defining a physics-based loss function (17) is a more obscure task, as mentioned at the beginning of this section. Therefore, here we provide a Python code snippet in Listing 1, where we define the loss function (17). To compute the Laplacian $\nabla^2 v$, we rely on automatic differentiation (Martín Abadi et al. 2015; Paszke et al. 2017), which is not a trivial task. Namely, computing first derivatives is a highly-optimised task that is well represented in literature and machine learning softwares' documentations due to its pivotal role in backpropagation. On the other hand, computing higher derivatives is a far less common task which therefore lacks representation in the documentation and optimised algorithms. To compute the Hessian matrix, we must first compute the gradient of the scalar function v with respect to the input tensor (x, y), which is a vector formed by the partial derivatives: $\nabla v = (v_x, v_y)$. Finally, the Hessian matrix $\mathbf{H}(v) = (v_{xx}, v_{xy}; v_{xy}, v_{yy})$ is obtained by taking the Jacobian of the gradient ∇v. Computing the Hessian can become a computationally demanding task for large models, so we use `tensorflow.GradientTape().batch_jacobian` instead of `tensorflow.GradientTape().jacobian` for the Jacobian computation. This is a much more efficient operation, but requires taking care that the gradients for every target-source set are independent (Martín Abadi et al. 2015).

```python
import tensorflow as tf

def pde_poisson(inputs, NN):
    x, y = tf.split(inputs, 2, axis=1)
    G = (1-x)*y**3 + x*(1+y**3)*tf.exp(-1.) + (1-y)*x*(tf.exp(-x) - tf.exp(-1.)) +
        y*((1+x)*tf.exp(-x) - (1-x+2*x*tf.exp(-1.))) # Satisfies Dirichlet b.cns

    with tf.GradientTape() as tape1:
        with tf.GradientTape() as tape2:.
            tape1.watch(inputs, x, y)
            tape2.watch(inputs, x, y)
            v = G + x*(x-1)*y*(y-1) * NN(inputs) # Trial solution
        d_v = tape2.gradient(v, inputs) # gradient of v w.r.t. inputs x,y: (v_x, v_y)
    d2_v = tape1.batch_jacobian(d_v, inputs) # Hessian matrices (v_xx, v_xy; v_yx, v_yy)

    v_xx = d2_v[:, 0, 0:1] # Extract (0,0) elements of the Hessian matrices
    v_yy = d2_v[:, 1, 1:2] # Extract (1,1) elements of the Hessian matrices
    f = -tf.exp(-x) * (-2+x+6*y+y**3)

    return tf.reduce_mean(tf.square(-v_xx - v_yy - f))
```

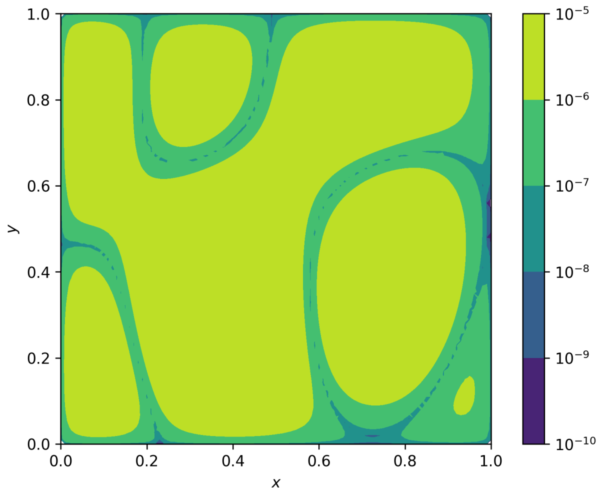
Listing 1: Defining loss function (17) in Tensorflow.

Finally, it remains to define and train the neural network. Following the original paper, we use a multilayer perceptron consisting of a single hidden layer with 10 hidden nodes and a sigmoid activation between the input and hidden layer, and a linear activation for the output layer. We use $n = 100$ input points (x, y) for training, and 900 points for testing, which were randomly sampled from a uniform distribution. To train the neural network, we use use Adam optimisation algorithm implemented in Tensorflow, with default hyperparameter values (learning rate $\eta = 0.001$, decay rates $\beta_1 = 0.9$, $\beta_2 = 0.999$).

After 20 000 training epochs, which took 7.5 seconds on a standard laptop, the model has achieved a training loss $-\nabla v - f = \mathcal{O}(10^{-7})$, and the final solution v is shown in Fig. 1a. Evaluating the model on the testing dataset of 900 previously unseen points yields a testing loss of the same order of magnitude. Knowing the analytical solution u, we compute the absolute error $|u - v|$, as shown in Fig. 1b. The deviation of the trial solution from the analytical solution is on the orders of 10^{-6} and 10^{-7} throughout most of the domain, while boundary conditions are satisfied exactly, as expected. The original paper Lagaris, Likas, and Fotiadis (1998) reports the absolute error on the order of 10^{-7} throughout the domain, except at the boundaries. With the low training loss and absolute error, we conclude that the model has converged to the sought solution, and that our implementation appears correct. Furthermore, the testing loss of the same order of magnitude as the training loss points to the model indeed having *learnt* the relation $-\nabla v = f$, and not having merely fit to the input data.



(a) $v = G + x(x-1)y(y-1)NN$, where G is given in (16).



(b) Absolute error $|u - v|$ between the trial solution v and analytical solution u.

Figure 1: (a) Trial solution v of Poisson problem (15) and (b) its accuracy at previously unseen points.

As a further validation step, we solved Poisson problem (15) using DeepXDE (Lu et al. 2021), whose solution matched our solution in Fig. 1. Furthermore, we searched GitHub for other implementations of the Lagaris, Likas, and Fotiadis (1998) method. Two implementations were found, which formed part of the authors' Master's dissertations. Both dissertations obtained seemingly correct solutions to problem (15). However, their implementation differs from the one presented here in the computation of the second derivatives in the Laplacian operator $\nabla^2(\cdot)$. Namely, the Tensorflow implementation by Paweł (2019) computed the second derivatives by taking the gradient of the gradient of v with respect to the input tensor (x, y). The training and testing loss is not reported, but the obtained absolute error is $|u - v| = \mathcal{O}(10^{-5})$ (Paweł 2019). The same computation of the second derivatives was performed in Hayden (2023) using PyTorch, which yielded a final training loss on the order of 10^{-7} for an optimal choice of the learning rate and an absolute error on the order of 10^{-6}.

Motivated by discovering this difference in implementations, we computed the second derivatives in the same fashion, as shown in the code snippet in Listing C.1. After 25 000 training epochs, the training loss has reached $\mathcal{O}(10^{-3})$ and the obtained trial solution is shown in Fig. C.1a. Visually, the solution does appear to be similar to the one shown in Fig. 1a. However, evaluating the model on the 900 previously unseen testing points yields a testing loss two orders of magnitude greater than the testing loss. Furthermore, the absolute error, shown in Fig. C.1b, is on the orders of 10^{-1} and 10^{-2} in most of the domain.

As noted earlier, the gradient of the trial solution v with respect to the input tensor (x, y) yields a nested structure of two tensors: $\nabla v = (v_x, v_y)$. By taking the gradient of this tensor structure ∇v with respect to the tensor (x, y), there is an implicit summation operation over the indices of the gradient tensor ∇v. That is, computing the gradient twice, as shown in Listing C.1, results in the following tensor:

$$\texttt{gradient[i,j]} = \sum_{k=1}^{n} \sum_{l=1}^{2} \frac{\partial\, \texttt{v[k,l]}}{\partial\, \texttt{inputs[i,j]}}, \tag{18}$$

whose shape is $(n, 2)$. Therefore, the loss function defined in Listing C.1 indeed does not match the Poisson problem (15), but instead describes

$$-(u_{xx} + 2u_{xy} + u_{xy}) = -e^{-x}(-2 + x + 6y + y^3). \tag{19}$$

The low training loss and seemingly correct solution obtained (Fig. C.1a) alone could give false confidence in one's implementation of the neural network model. However, the importance of evaluating the model on previously unseen data cannot be understated. Indeed, the high absolute error (Fig. C.1b) and testing loss two orders of magnitudes higher than the training loss indicate that the model has *not learnt* the correct physical relation, but is instead suffering from overfitting the training data.

Finally, we are confident that the implementation presented in this section and Listing 1 is indeed correct. In the next section we therefore apply the same implementation on a new problem, where we will instead focus on finding optimal model and training configurations.

3 Results and analysis

In this section, we again consider the Poisson problem on a unit square domain with homogeneous Dirichlet boundary conditions defined in eq. (1). In particular, by choosing the source term $f = 13\pi^2 \sin(2\pi x)\sin(3\pi y)$, the Poisson problem in question is

$$\begin{aligned} -\nabla^2 u &= 13\pi^2 \sin(2\pi x)\sin(3\pi y) \qquad && \text{in } \Omega = (0,1)^2, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned} \tag{20}$$

The data f was obtained by taking the Laplacian of the analytical solution $u = \sin(2\pi x)\sin(3\pi y)$, which is shown in Fig. 2.

We begin by explaining the motivation behind choosing the mentioned solution that is to be approximated, which can be separated into the following points:

1. Poisson problem with homogeneous boundary conditions (eq. (1)) is straightforward to discretise using finite-difference methods, as outlined in section 2.1;
2. The convergence of basic iterative methods displays interesting properties under certain conditions, which we show in section 3.1;
3. While neural networks can approximate any continuous function (Hornik, Stinchcombe, and White 1989), they do not necessarily do so efficiently. Namely, neural networks are notoriously inefficient at approximating periodic functions (Ziyin, Hartwig, and Ueda 2020; Dong and Ni 2021). A Google search quickly reveals a plethora of questions submitted on machine learning forums asking for help with approximating such functions. Existing literature offers limited clues on why this is the case, and

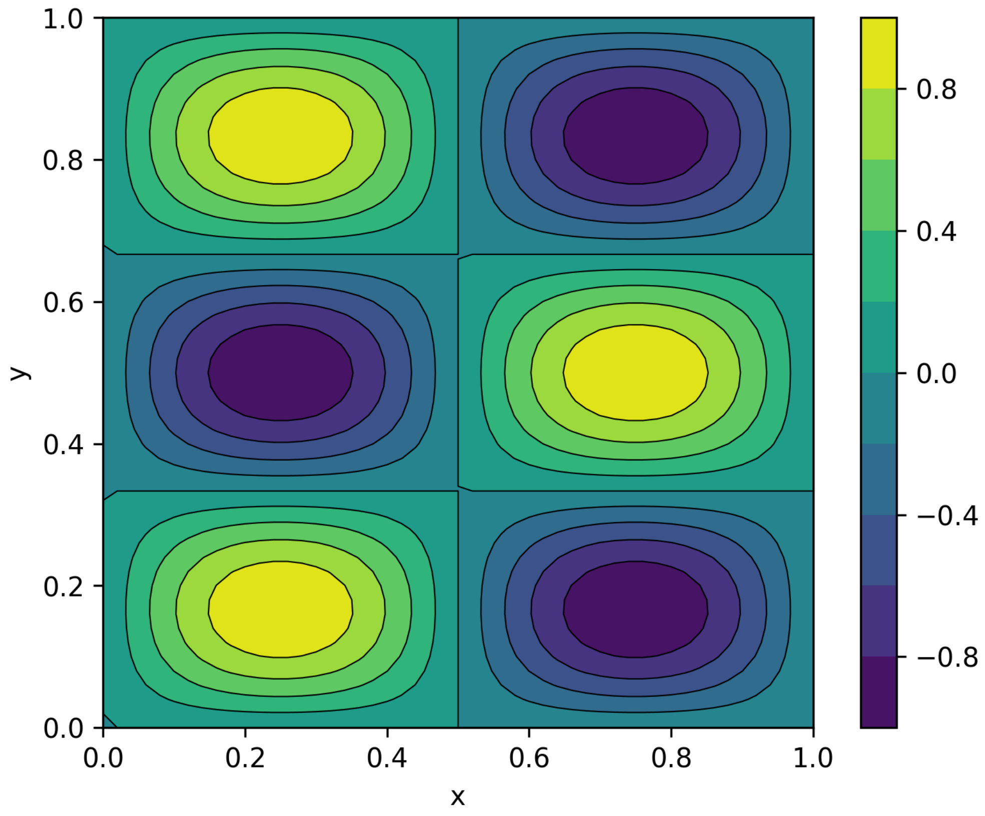


Figure 2: Analytical solution $u = \sin(2\pi x)\sin(3\pi y)$ of the Poisson problem (20).

it remains an active research question (Ziyin, Hartwig, and Ueda 2020). By choosing a periodic target function, we introduce added complexity to our model and in finding its optimal performance;

4. Neural network models with a correctly defined loss function (such as in Listing 1, Fig. 4) yield a clearly different result to the solution obtained by models with an incorrectly defined loss function (such as in Listing C.1, Fig. C.2).

In the next two subsections we solve the Poisson problem (20) using two approaches. In section 3.1 we find the solution using Jacobi and Gauss-Seidel iterative methods, as representatives of traditional numerical methods with rigorous theoretical underpinning that is expected in scientific computing. We compare these to finding the solution using the PINN method of Lagaris, Likas, and Fotiadis (1998) in section 3.2, as a representation of a modern data-driven technique.

3.1 Basic iterative methods

Finding the solution to eq. (20) using Jacobi and Gauss-Seidel iterative methods requires transforming the problem to a discretised form, suitable to be solved on a computer. We discretise the problem (20) on a uniform grid of 50×50 points using a finite difference method outlined in section 2.1, which yields equal step sizes in both directions $\Delta x = \Delta y = 1/50$. Therefore, we obtain a large sparse system of 49×49 equations $A\mathbf{U} = \mathbf{f}$ (2) for the interior of the domain. To take advantage of the sparse structure of the matrix A, we use the `SciPy.sparse` python package for more efficient performance (Virtanen et al. 2020). Furthermore, we use the in-built `scipy.sparse.linalg.spsolve` function to solve the sparse system $A\hat{\mathbf{U}} = \mathbf{f}$ for the *exact* numerical solution $\hat{\mathbf{U}}$.

Now we wish to solve the system (2) using our own implementation of the Jacobi and Gauss-Seidel methods, as described in section 2.1, to obtain the solution $\mathbf{U}$ which we shall compare to the exact discretised solution $\hat{\mathbf{U}}$. Iterative methods require us to choose an initial guess $\mathbf{U}^{(0)}$. The rigorous theoretical underpinning of iterative methods allows us to cleverly choose the initial guess $\mathbf{U}^{(0)}$ in order to optimise the convergence. Therefore, we perform two experiments: in the first experiment we choose an initial guess to be $\mathbf{U}^{(0)} = \mathbf{0}$ and in the second we initialise $\mathbf{U}^{(0)}$ with random small elements centred around zero. The convergence of the Jacobi and Gauss-Seidel methods in both experiments are shown in Fig. 3.

The motivation for the two sets of experiments becomes more apparent by observing the difference of convergence rates between them. For the initial guess $\mathbf{U}^{(0)} = \mathbf{0}$, the residual $\|\mathbf{U}^{(k)} - \hat{\mathbf{U}}\| = 10^{-10}$ has been reached faster for Jacobi iteration than for the Gauss-Seidel iteration, shown as darker-coloured lines in Fig 3. This is an interesting observation, since we know that Gauss-Seidel should be faster due to a larger spectral radius $\rho(C_{GS}) = (\rho(C_J))^2$. However, in the case of $\mathbf{U}^{(0)} = \mathbf{0}$, the next two Jacobi iterates are $\mathbf{U}^{(1)} = D^{-1}\mathbf{f} = (2/\Delta x^2 + 2/\Delta y^2)\mathbf{f}$ and $\mathbf{U}^{(2)} = (2/\Delta x^2 + 2/\Delta y^2)\ C_J\mathbf{f} + D^{-1}\mathbf{f}$. Namely, by studying the spectral properties of the matrix A (see appendix A), we observe that the solution $\mathbf{U}^{(k)}$ and the error $\mathbf{e}^{(k)}$ are eigenvectors of the Jacobi iteration matrix corresponding to the eigenvalue

$$\lambda = \frac{1}{2}\cos(2\pi\Delta x)\cos(3\pi\Delta x), \tag{21}$$

which is below the spectral radius $\rho(C_J)$. Therefore, the error gets removed very quickly in Jacobi iterations. Meanwhile, the error in the Gauss-Seidel iterations initially decays at the same rate, before convergence starts

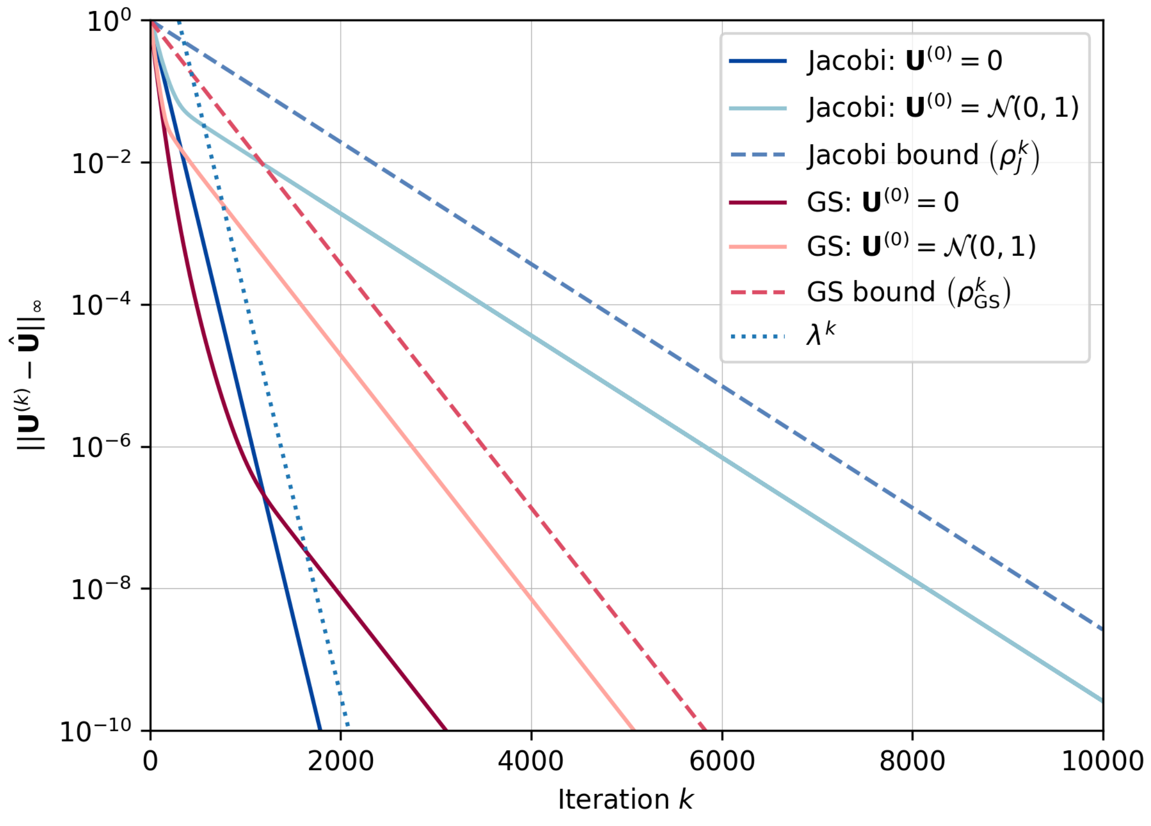


Figure 3: Convergence of the error $\mathbf{e}^{(k)}$ with Jacobi and Gauss-Seidel (GS) iterates applied to the discretised Poisson problem (2) for initial guess $\mathbf{U}^{(0)} = \mathbf{0}$ (dark blue and dark red lines) and $\mathbf{U}^{(0)} = \mathcal{N}(0, 1)$ (light blue and light red lines) on a 50×50 uniform grid. Theoretical convergence bounds corresponding to the spectral radii are shown with dashed lines. Convergence rate corresponding to the eigenvalue in eq. (21) is shown with a dotted line.

slowing down to be governed by $\rho(C_{GS})$ and finally becoming parallel with the convergence rate corresponding to it, shown as a dashed line in Fig. 3.

In the next experiment we initialise $\mathbf{U}^{(0)}$ with random small elements centred around zero and again solve eq. (2) using Jacobi and Gauss-Seidel methods. Convergence rates are shown in Fig. 3 with lightly-coloured lines. Indeed, we now observe the expected convergence rates since the error is no longer a component of a single eigenvector, but becomes a component of multiple eigenvectors. In particular, this includes the one associated with $\rho(C_J) = \lambda(C_J)^{(N-1,N-1)}$ which comes to dominate the error term for Jacobi, and $\rho(C_{GS}) = (\rho(C_J))^2$ for the Gauss-Seidel iterates, shown as dashed lines in Fig. 3.

3.2 Solution method involving a neural network

Finally, it remains to solve the Poisson problem (20) using the Lagaris, Likas, and Fotiadis (1998) PINN method. Given homogeneous boundary conditions, we choose the function G to equal zero, leading to the trial solution function v to be

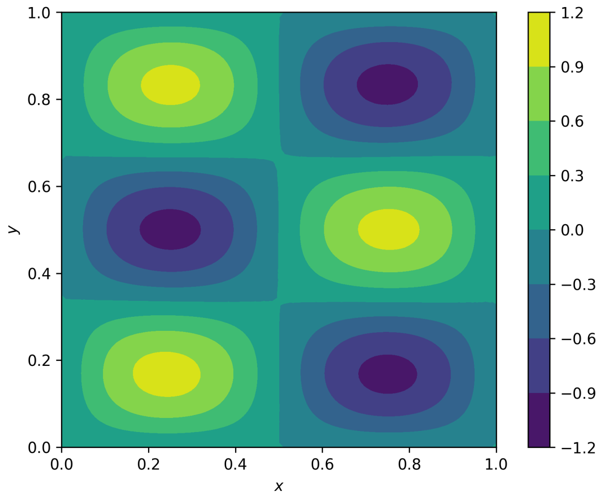
$$v = x(x-1)y(y-1)NN. \tag{22}$$

The definition of the loss function remains largely the same as that shown in Listing 1, where the only necessary changes are the definition of `G = 0` in Line 5 and `f = 13*np.pi**2 * tf.sin(2*np.pi*x) * tf.sin(3*np.pi*y)` in Line 17. In all experiments that follow, we again sample 1000 points in the domain from a uniform distribution, which we split into a training set of 100 points, and a testing set of 900 points.

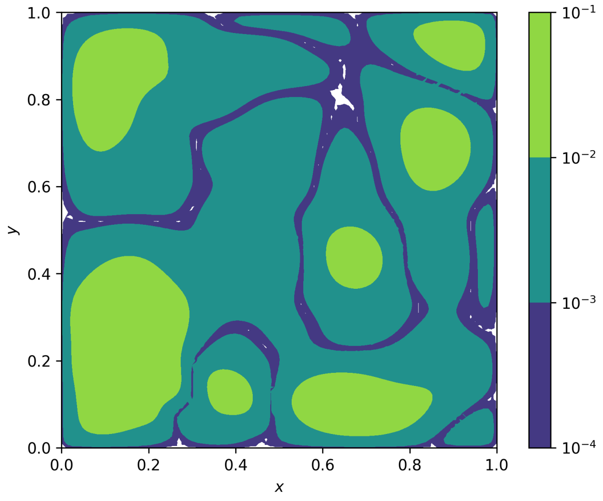
We begin by naively trying a simple PINN architecture consisting of a single hidden layer of 16 nodes with a hyperbolic tangent (tanh) activation, and a single node-output layer with a linear activation. We train the network for 50 000 epochs using an Adam optimiser with default hyperparameter values (learning rate of 0.001, $\beta_1 = 0.9, \beta_2 = 0.999$). The model achieved a training loss $\mathcal{O}(10^0)$ and a test loss $\mathcal{O}(10^1)$. The final trial solution (22) is shown in Fig. 4a and the absolute error in Fig. 4b. We observe that the trial function v deviates from the analytical solution u by a factor of 10^{-2} and 10^{-3} in most of the domain, while the boundaries are again satisfied exactly. As expected, this presents a more challenging problem than that considered in section 2.2.1.

Therefore, we perform several experiments in an attempt to find more optimal performance. We vary the learning rate $\eta \in \{10^{-2}, 10^{-3}, 10^{-4}\}$, the number of hidden layers between 1 and 3, and the number of hidden nodes in each layer to be either 16 or 32. We also test the impact of different activation functions. Since we require the activation function to be twice differentiable, we consider the $\tanh(x)$ and $\text{sigmoid}(x)$ activation functions, as well as the activation function $x + \sin^2(x)$ which was proposed in the literature to be superior in learning periodic functions (Ziyin, Hartwig, and Ueda 2020). This results in 54 different configurations. Each experiment uses the same 100 training and 900 testing points, and equal number of 50 000 training epochs.

The results of the parameter study are shown in three parallel-coordinates style plot in Fig. 5 and are sorted in the order of increasing testing loss in Table D.0. We observe that the number of hidden layers appears to be the most impactful parameter on the model performance, where models with a single hidden layer achieved the lowest testing loss. Networks comprising a single hidden layer with the $x + \sin^2(x)$ activation function in particular perform the best. These networks achieved the testing loss 1 to 3 orders of magnitude lower than that of the next best performing model, with the exception of a single-layered neural network with 16 hidden units trained with a low learning rate $\eta = 10^{-4}$. The slow learning rate might have caused the function to converge to a local minimum or might simply require more training epochs. The next best-performing single-layer models are those with 16 hidden nodes and a $\text{sigmoid}(x)$ activation function, whose



(a) Trial solution $v = x(x-1)y(y-1)NN$.



(b) Absolute error $|u - v|$ between the trial solution v and analytical solution u.

Figure 4: (a) Trial solution v of Poisson problem (22) and (b) its accuracy at previously unseen points for a PINN consisting of 1 hidden layer with 16 nodes, trained for 50 000 epochs using the Adam optimiser with default hyperparameter values.

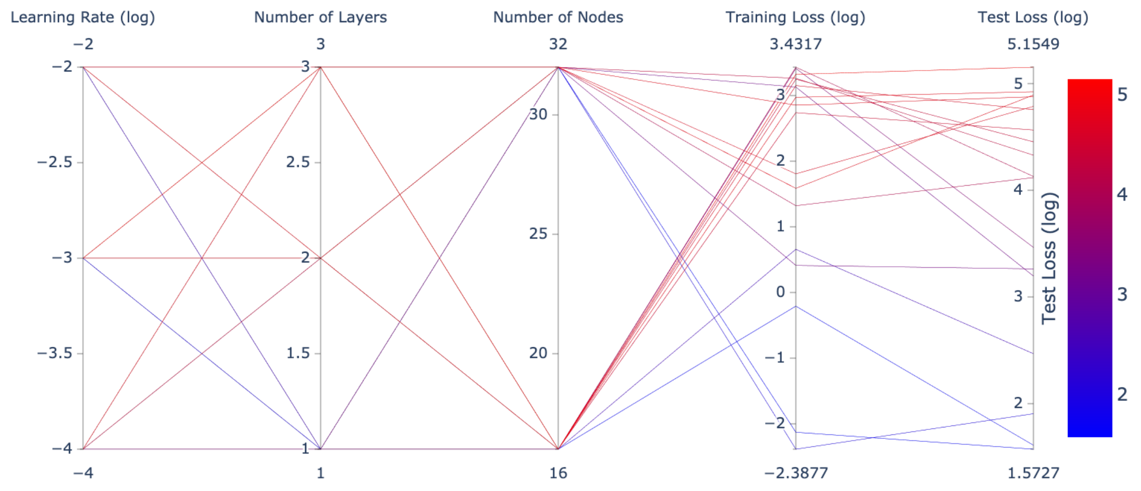
testing loss is $\mathcal{O}(10^1)$, while those with 32 hidden nodes have a testing loss that is 4 orders of magnitude higher than their training loss, indicating that they are overfitting the training data rather than learning the physical relation (17). Similar indications of overfitting are found by looking at the single-layered models with the $\tanh(x)$ activation function and learning rates 10^{-2} and 10^{-3}. All models comprising 2 or 3 hidden layers with $\tanh(x)$ and $\text{sigmoid}(x)$ activation function have both training and testing losses high, with the exception of a 3×32 architecture with a $\text{sigmoid}(x)$ activation which appears to overfit the training data. On the other hands, all multi-layer models with $x + \sin^2(x)$ activation have 3 to 10 orders of magnitude higher testing loss than training loss.

In summary, we observe that the lowest testing loss $\mathcal{O}(10^{-2})$ was achieved using the $x + \sin^2(x)$ activation function for a single hidden layer with 32 nodes and learning rates of 10^{-3} and 10^{-2}. The best performing model solution is shown in Fig. 6a and its accuracy in Fig. 6b, which is on the order of 10^{-4} and 10^{-5} throughout most of the domain.

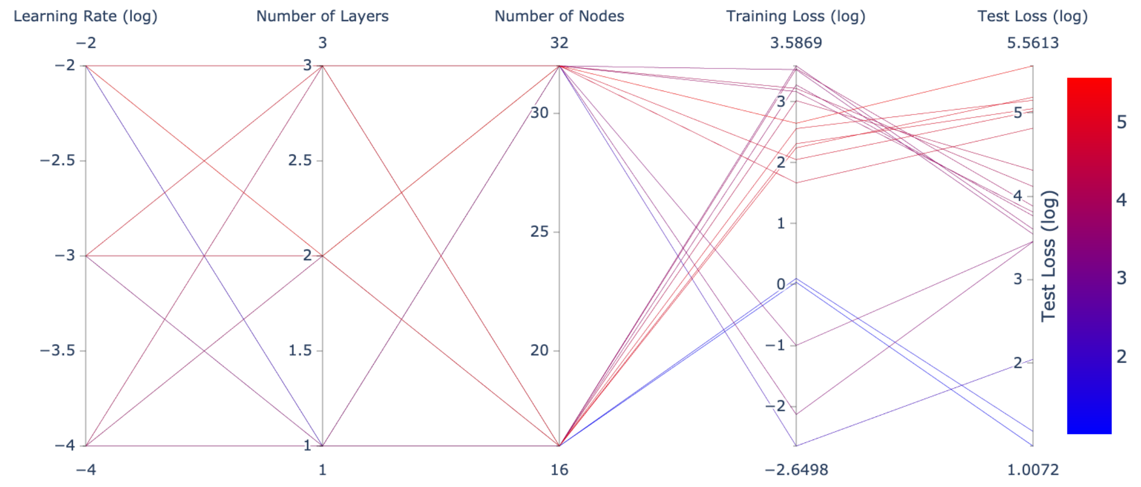
4 Conclusions

In this project we have implemented two fundamentally different methods for solving Poisson's equation with Dirichlet boundary conditions. First we discretised the Poisson problem using finite differences to obtain a large sparse system of linear equations and then implemented the Jacobi and Gauss-Seidel iterative schemes to solve it. These are prominent techniques in numerical analysis, which iteratively converge to the sought solution under conditions and with a convergence rate that can be analytically studied and leveraged upon to achieve optimal convergence (Fig. 3).

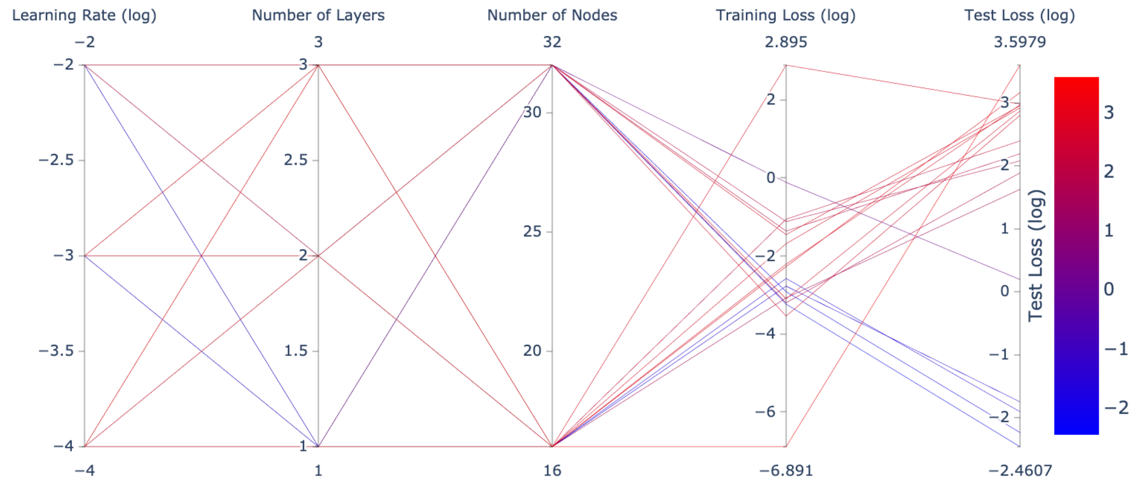
Afterwards, a specific method involving physics-informed neural networks (PINNs) was implemented, where neural networks were trained to learn the Poisson's Partial Differential Equation relation defined in the loss function. In particular, the method by Lagaris, Likas, and Fotiadis (1998) was implemented in the Tensorflow machine learning library, where we seek a trial solution that satisfies the boundary conditions by construction (section 2.2). A large part of this project was dedicated to ensuring the validity of our



(a) $\tanh(x)$

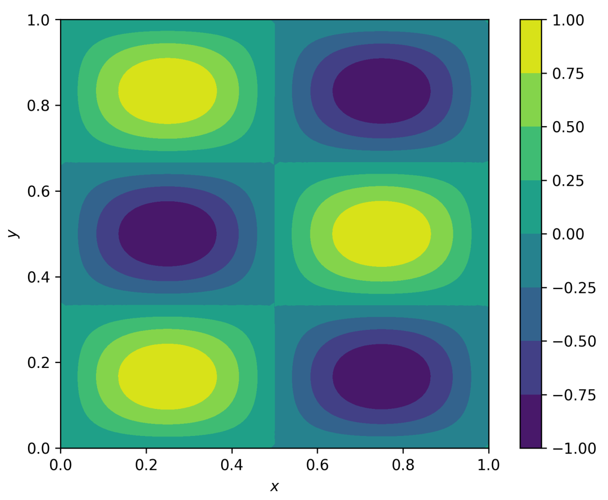


(b) $\text{sigmoid}(x)$

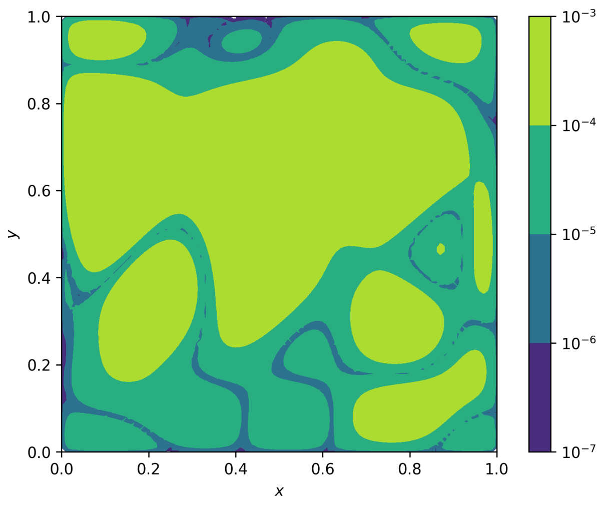


(c) $x + \sin^2(x)$

Figure 5: Test loss (17) on 900 previously unseen points for 54 different neural networks trained for 50 000 epochs with varying learning rate of the Adam optimiser, number of hidden layers, number of nodes in each hidden layer and different activation functions: (a) $\tanh(x)$, (b) $\text{sigmoid}(x)$, and (c) $x + \sin^2(x)$, proposed by Ziyin, Hartwig, and Ueda (2020).



(a) Trial solution $v = x(x-1)y(y-1)NN$.



(b) Absolute error $|u - v|$ between the trial solution v and analytical solution u.

Figure 6: (a) Trial solution v of Poisson problem (22) and (b) its accuracy at previously unseen points for a PINN consisting of 1 hidden layer with 32 nodes, activation function $x + \sin^2(x)$, trained for 50 000 epochs using the Adam optimiser with a learning rate of 10^{-3}.

implementation of the method (section 2.2.1). It was found that a PINN with an incorrectly defined loss function may still lead to a decently accurate solution. In particular, this was the case when we considered Poisson's eq. (15) with analytical solution $u_1 = e^{-x}(x + y^3)$ (Fig. 1 and Fig. C.1). Function u_1 is a smooth function, as it exhibits gradual changes without abrupt oscillations or rapid fluctuations, as it is characterised by exponential decay e^{-x} in the x-direction and involves polynomial terms $(x + y^3)$ which generally result in smooth variations. On the other hand, this was not observed when considering Poisson's eq. (20) (Fig. 4 and Fig. C.2) whose analytical solution is $u_2 = \sin(2\pi x)\sin(3\pi y)$. Function u_2 is an oscillatory function due to the product of sine functions which results in rapid changes and multiple peaks and troughs, making it less smooth compared to u_1. By defining the trial solution that automatically satisfies boundary conditions, we believe that this boundary term can dominate the interior of a function such as u_1 that varies gradually away from the boundaries, as opposed to a more oscillatory function like u_2. In such case, the model is still able to converge to a low training loss, but retains a high loss when evaluated on previously unseen input points (Fig. C.1). This indicates that the model is overfitting the training data, rather than learning the underlying physical principle.

Finally, we varied the neural network and training configurations in order to find the set of parameters leading to the optimal model performance. The problem considered was Poisson's eq. (20) with solution u_2, chosen in part because neural networks are notoriously inefficient at approximating period functions and require careful consideration, which remains an active research question. To overcome said complexity of approximating periodic functions, we consider the activation function $x + \sin^2(x)$, which achieves the periodic inductive bias that neural networks lack (Ziyin, Hartwig, and Ueda 2020). Indeed, models involving $x + \sin^2(x)$ activation far outperform models with sigmoid(x) and $\tanh(x)$ activation between hidden layers (Fig. 5, Table D.0). Furthermore, models comprising only a single hidden layer outperform multi-layer models, which tend to overfit the training data. This indicates that a single hidden layer has sufficient representational capacity to capture the frequency of the simple oscillatory pattern of u_2.

It must be stressed that this project did not utilise the full flexibility that neural networks and their training algorithms offer. The parameter study employed here is far from being extensive, and could have included, for example, different functions to initialise neural network parameters or varying the choice of optimiser.

However, it is unclear whether these would provide more optimal performance than that found in this report. Indeed, the machine learning research is still relatively novel, and often lacks the theoretical rigour that is required in most scientific and engineering disciplines, and that traditional numerical methods offer, such as Jacobi and Gauss-Seidel methods presented here. Therfore, one of the most common critiques of deep learning models is that they are mostly designed by trial and error, and where solutions produced might appear correct, but are indeed wrong, such as was shown in section 2.2.1. However, the field of physics-informed machine learning is rapidly growing, and further theoretical improvements are expected to be made.

Code availability

This report is accompanied by the source code for the demonstrations used in this report. The code comprises:

- `model.py`: A python script containing the base class for all neural network-based models used in this report.
- `plot_utilities.py`: A convenience python script containing functions used for plotting.
- `results_lagaris_ex5.ipynb`: A Jupyter notebook demonstrating the solution of Poisson problem (15) using Lagaris, Likas, and Fotiadis (1998) method as described in section 2.2. Produces Figures 1 and C.1.
- `results_iterative.ipynb`: A Jupyter notebook demonstrating the solution of Poisson problem (20) using Jacobi and Gauss-Seidel iteration methods, as described in sections 2.1 and 3.1. Produces Figures 2 and 3 and Table D.0.
- `results_lagaris.ipynb`: A Jupyter notebook demonstrating the solution of Poisson problem (20) using Lagaris, Likas, and Fotiadis (1998) method, as described in sections 2.2 and 3.2. Produces Figures 4, 5, 6 and C.2.

References

Blechschmidt, Jan, and Oliver G Ernst. 2021. “Three ways to solve partial differential equations with neural networks—A review.” *GAMM-Mitteilungen* 44 (2): e202100006.

Briggs, William L, Van Emden Henson, and Steve F McCormick. 2000. *A multigrid tutorial.* SIAM.

Demmel, James W. 1997. *Applied numerical linear algebra.* SIAM.

Dong, Suchuan, and Naxian Ni. 2021. “A method for representing periodic functions and enforcing exactly periodic boundary conditions with deep neural networks.” *Journal of Computational Physics* 435:110242.

Greenfeld, Daniel, Meirav Galun, Ronen Basri, Irad Yavneh, and Ron Kimmel. 2019. “Learning to optimize multigrid PDE solvers.” In *International Conference on Machine Learning,* 2415–2423. PMLR.

Hayden, Isaac S. 2023. *Neural Networks for Solving Differential Equations.* Master’s thesis, University of Durham. https://github.com/Isaac-Somerville/Neural-Networks-for-Solving-Differential-Equations/tree/main.

Hjorth-Jensen, Morten, and Sebastian Gregorius Winther-Larsen. n.d. *Compphysics/ComputationalPhysics: Introductory course in computational physics, including linear algebra, eigenvalue problems, differential equations, Monte Carlo methods and more.* https://github.com/CompPhysics/ComputationalPhysics.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. 1989. “Multilayer feedforward networks are universal approximators.” *Neural networks* 2 (5): 359–366.

Iserles, Arieh. 2009. *A first course in the numerical analysis of differential equations.* 44. Cambridge university press.

Karniadakis, George Em, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. 2021. “Physics-informed machine learning.” *Nature Reviews Physics* 3 (6): 422–440.

Lagaris, Isaac E, Aristidis Likas, and Dimitrios I Fotiadis. 1998. “Artificial neural networks for solving ordinary and partial differential equations.” *IEEE transactions on neural networks* 9 (5): 987–1000.

LeVeque, Randall J. 2007. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems.* SIAM.

Lu, Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. 2021. “DeepXDE: A deep learning library for solving differential equations.” *SIAM Review* 63 (1): 208–228. https://doi.org/10.1137/19M1274067.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. https://www.tensorflow.org/.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. “Automatic differentiation in pytorch.”

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. “Pytorch: An imperative style, high-performance deep learning library.” *Advances in neural information processing systems* 32.

Paweł, Grabiński. 2019. *Numerical Solution for Differential Equations with Neural Networks.* Master’s thesis, University of Wroclaw. https://github.com/PGrabinski/NeuralDifferentialEquations.

Pereira, Danilo R, Marco Antonio Piteri, André N Souza, João Paulo Papa, and Hojjat Adeli. 2020. “FEMa: A finite element machine for fast learning.” *Neural Computing and Applications* 32:6393–6404.

Pérez, Patrick, Michel Gangnet, and Andrew Blake. 2003. "Poisson Image Editing." *ACM Trans. Graph.* (New York, NY, USA) 22, no. 3 (July): 313–318. ISSN: 0730-0301. https://doi.org/10.1145/882262.882269. https://doi.org/10.1145/882262.882269.

———. 2023. "Poisson Image Editing." In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2,* 1st ed. New York, NY, USA: Association for Computing Machinery. ISBN: 9798400708978. https://doi.org/10.1145/3596711.3596772.

Rackauckas, Chris, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. 2019. "Diffeqflux.jl-A julia library for neural differential equations." *arXiv preprint arXiv:1902.02376.*

Rackauckas, Christopher, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. 2020. "Universal differential equations for scientific machine learning." *arXiv preprint arXiv:2001.04385.*

Raissi, Maziar, Paris Perdikaris, and George E Karniadakis. 2019. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." *Journal of Computational physics* 378:686–707.

Trefethen, Lloyd N, and David Bau. 1997. *Numerical linear algebra.* Vol. 181. Siam.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, et al. 2020. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods* 17:261–272. https://doi.org/10.1038/s41592-019-0686-2.

Ziyin, Liu, Tilman Hartwig, and Masahito Ueda. 2020. "Neural networks fail to learn periodic functions and how to fix it." *Advances in Neural Information Processing Systems* 33:1583–1594.

Zubov, Kirill, Zoe McCarthy, Yingbo Ma, Francesco Calisto, Valerio Pagliarino, Simone Azeglio, Luca Bottero, et al. 2021. *NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations.* https://doi.org/10.48550/ARXIV.2107.09443. https://arxiv.org/abs/2107.09443.

Appendix A Spectral properties

Written in the form (7), it is easy to show that the eigenvalues of A are a sum of the eigenvalues of the two matrices written in full in (7)

$$\lambda(A) = \lambda^{(r,s)} = \lambda_r + \lambda_s = 2\left(\frac{1-\cos(r\pi\Delta x)}{\Delta x^2} + \frac{1-\cos(s\pi\Delta y)}{\Delta y^2}\right) \tag{A.1}$$

with corresponding eigenvectors

$$V_{i,j}^{(r,s)} = \sin(ri\pi\Delta x)\sin(sj\pi\Delta y) \tag{A.2}$$

for $1 \leq r, i \leq N-1$, $1 \leq s, j \leq M-1$ (Demmel 1997).

Given that we can write $D = (2/\Delta x^2 + 2/\Delta y^2)I$ and $L + R = A - D$, we can rewrite the Jacobi iteration matrix $C_J = -D^{-1}(L+R)$ as

$$\begin{aligned} C_J &= -D^{-1}(L+R) \\ &= -\left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right)^{-1} I\left(A - \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right)I\right) \\ &= I - \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right)^{-1} IA. \end{aligned} \tag{A.3}$$

Therefore, the eigenvalues of C_J are given by

$$\begin{aligned} \lambda(C_J)^{(r,s)} &= 1 - \left(\frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}\right)^{-1} \lambda^{(r,s)} \\ &= \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1}\left(\frac{\cos(r\pi\Delta x)}{\Delta x^2} + \frac{\cos(s\pi\Delta y)}{\Delta y^2}\right), \end{aligned} \tag{A.4}$$

where $\lambda^{(r,s)}$ were given in eq. (A.1).

Appendix B Sign error in Lagaris et al. (1998)

As claimed in section 2.2.1, the function $G(x,y)$ chosen by the original authors of the method (Lagaris, Likas, and Fotiadis 1998) does not satisfy the given boundary condition $u(x,1) = e^{-x}(x+1)$. The sign error appears in the final term of the authors' chosen function:

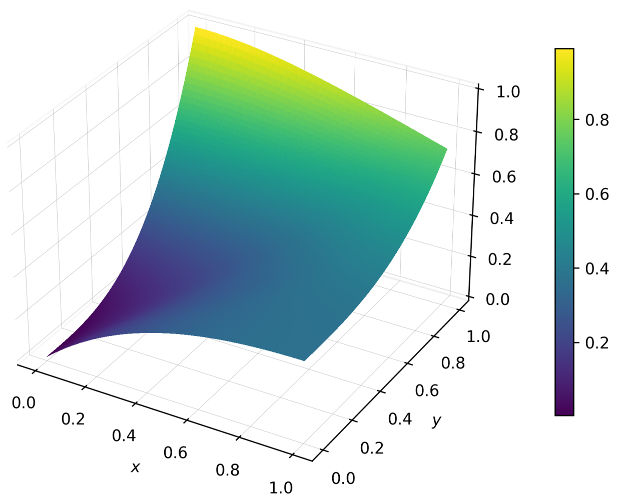
$$G = (1-x)y^3 + x(1+y^3)e^{-1} + (1-y)x(e^{-x} - e^{-1}) + y((1+x)e^{-x} - (1 - x - 2xe^{-1})). \tag{B.1}$$

Indeed, $G(x,1) = e^{-x}(1+x) + 4xe^{-1}$ does not satisfy the boundary condition $u(x,1) = e^{-x}(x+1)$. The sought boundary conditions are satisfied by changing the sign of the final term, as in eq. (16), which yields $G(x,1) = e^{-x}(1+x)$.

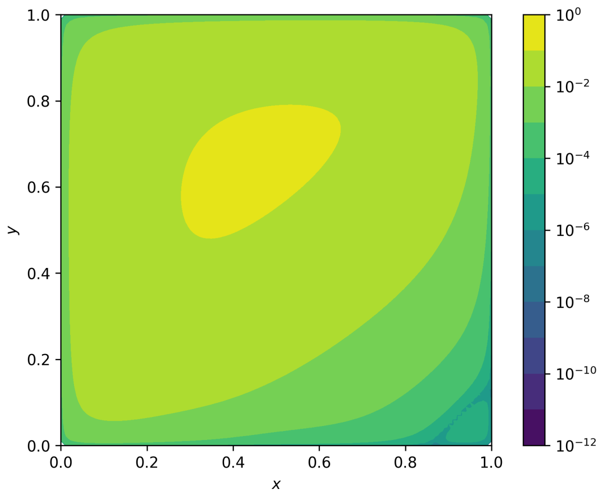
Appendix C Incorrectly computed second derivatives

```python
import tensorflow as tf

def pde_poisson(inputs, nn):
    x, y = tf.split(inputs, 2, axis=1)
    G = (1-x)*y**3 + x*(1+y**3)*tf.exp(-1.) + (1-y)*x*(tf.exp(-x) - tf.exp(-1.)) +
        y*((1+x)*tf.exp(-x) - (1-x+2*x*tf.exp(-1.))) # Satisfies Dirichlet b.cns

    with tf.GradientTape() as tape1:
        with tf.GradientTape() as tape2:.
            tape1.watch(inputs, x, y)
            tape2.watch(inputs, x, y)
            v = G + x*(x-1)*y*(y-1) * nn(inputs) # Trial solution
        d_v = tape2.gradient(v, inputs) # gradient of v w.r.t. inputs x,y: (v_x, v_y)
    d2_v = tape1.gradient(d_v, x) # THIS IS WRONG! Implicit summation: (v_xx + v_xy, v_yy + v_xy)

    v_xx = d2_v[:, 0:1] # THIS IS WRONG! This is: v_xx + v_xy
    v_yy = d2_v[:, 1:2] # THIS IS WRONG! This is: v_yy + v_xy
    f = -tf.exp(-x) * (-2+x+6*y+y**3)

    return tf.reduce_mean(tf.square(-v_xx - v_yy - f))
```

Listing C.1: Incorrectly defining loss function (17) in Tensorflow.

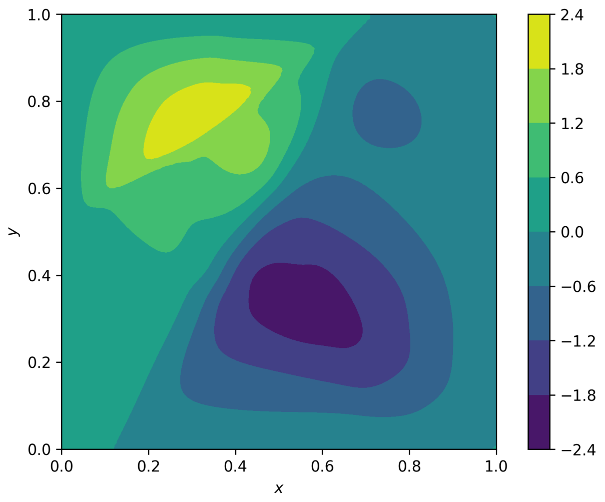


(a) Trial solution $v = G + x(x-1)y(y-1)NN$.

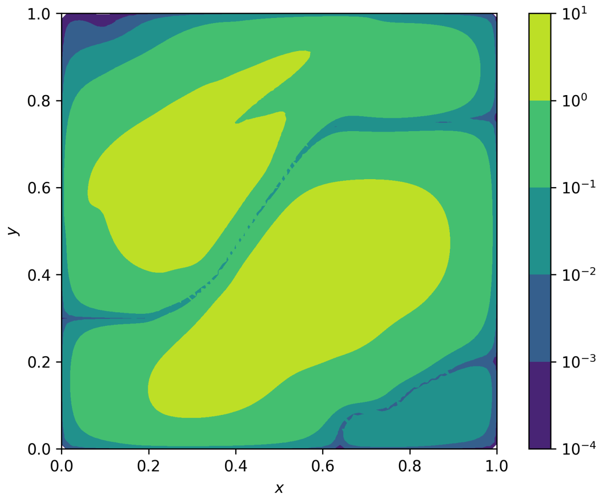


(b) Absolute error $|u - v|$ between the trial solution v and analytical solution u.

Figure C.1: (a) Trial solution v of Poisson problem (15) and (b) its accuracy at previously unseen points for incorrectly computed second derivatives in the loss function, as in Listing C.1 with the same network and training configuration as in Fig. 1.



(a) Trial solution $v = x(x-1)y(y-1)NN$.



(b) Absolute error $|u - v|$ between the trial solution v and analytical solution u.

Figure C.2: (a) Trial solution v of Poisson problem (20) and (b) its accuracy at previously unseen points for incorrectly computed second derivatives in the loss function, as in Listing C.1 with the same network and training configuration as in Fig. 4.

Appendix D Parameter study

learning˙rate	activation	num˙layers	num˙nodes	training˙loss	test˙loss
-3.0	x+sin$^2(x)$	1	32	-3.234215054936895	-2.4607415370387806
-2.0	x+sin$^2(x)$	1	32	-2.9155337253520046	-2.235940522868182
-3.0	x+sin$^2(x)$	1	16	-2.576882817709953	-1.9047788724722396
-2.0	x+sin$^2(x)$	1	16	-2.7724650975053255	-1.7486203331158654
-4.0	x+sin$^2(x)$	1	32	-0.1125191286714653	0.1939899108729626
-3.0	sigmoid	1	16	0.03833967714353205	1.0071581280503743
-2.0	sigmoid	1	16	0.09974226868672778	1.182885337653372
-2.0	tanh	1	32	-2.129790783811023	1.5726565678020723
-3.0	tanh	1	16	-0.20861152192071078	1.6096980649015349
-2.0	x+sin$^2(x)$	2	16	-3.0743192098767893	1.626060089236302
-3.0	x+sin$^2(x)$	2	32	-3.197528088303518	1.887972507486014
-3.0	tanh	1	32	-2.3877451060138513	1.9051692056601957
-2.0	sigmoid	1	32	-2.6497741522401834	2.046897223526237
-2.0	x+sin$^2(x)$	3	32	-1.1166965230103751	2.077576691690862
-2.0	x+sin$^2(x)$	2	32	-1.3604013011277092	2.1910645488877463
-2.0	x+sin$^2(x)$	3	16	-1.05528829194451	2.39474636748207
-2.0	tanh	1	16	0.6574712740438114	2.4678148041957737
-4.0	x+sin$^2(x)$	2	32	-3.54143873522092	2.801171464574732
-4.0	x+sin$^2(x)$	3	32	-3.0983954390580863	2.8702289210595344
-4.0	x+sin$^2(x)$	2	16	-1.683883361064575	2.9192712859420475
-3.0	x+sin$^2(x)$	3	32	-1.451512351847439	2.953288472819769
-3.0	x+sin$^2(x)$	2	16	-2.2091810324304992	2.9663461152156616
-4.0	x+sin$^2(x)$	1	16	2.894998649624144	2.9797250157002035
-3.0	x+sin$^2(x)$	3	16	-2.2724353198105245	3.162404135082693
-4.0	tanh	1	32	3.1292852107687428	3.1962142173289743
-2.0	tanh	3	32	0.41165640089689276	3.2635008024801615
-3.0	sigmoid	1	32	-2.1325742226438353	3.454674557000404
-2.0	sigmoid	3	32	-0.9992679373672251	3.4581020208652804
-4.0	tanh	1	16	3.4131731119750857	3.4649791911253143
-4.0	sigmoid	1	32	3.5242446675492114	3.5455897334446584
-4.0	x+sin$^2(x)$	3	16	-6.891155307017796	3.597892097562353
-4.0	sigmoid	1	16	3.586919078159338	3.604666529295721
-4.0	sigmoid	2	16	3.275033847126095	3.763959215186435
-4.0	sigmoid	2	32	3.164393468691547	3.8096682939209194
-4.0	sigmoid	3	16	3.5396386288505632	3.8807413535032778
-4.0	sigmoid	3	32	3.2172325514599978	4.116205157108664
-2.0	tanh	2	32	1.3212697822667923	4.11957639205962
-4.0	tanh	2	16	3.43173240643627	4.129994639584325
-3.0	sigmoid	2	16	3.016761581599175	4.310158013209839
-4.0	tanh	2	32	3.261790332683861	4.329608433940295
-4.0	tanh	3	16	3.2486894704109064	4.454594916459677
-3.0	tanh	2	16	2.739131257522273	4.565730709187698
-2.0	tanh	3	16	3.1517534714459545	4.756853141823065
-3.0	tanh	3	32	1.8063085015305465	4.78739842556386
-3.0	sigmoid	2	32	1.6669774453299016	4.814628495438818
-3.0	tanh	2	32	2.8533567271508127	4.877921797826325
-4.0	tanh	3	32	1.5865152034696057	4.893739646124301
-2.0	tanh	2	16	2.9733149529447713	4.924320448222361
-3.0	sigmoid	3	32	2.0490286319899127	5.019063047329675
-2.0	sigmoid	3	16	2.309881685544687	5.0493605124265635
-3.0	sigmoid	3	16	2.558076308816776	5.149436964271863
-3.0	tanh	3	16	3.322154358099832	5.154868809348987
-2.0	sigmoid	2	16	2.241913461747332	5.18627549254694
-2.0	sigmoid	2	32	2.645260684819938	5.561285333218871

Table D.0: Results of the parameter study described in section 3.2.