Numerical Solutions of Differential Equations using Neural Networks

Candidate Number: 1090497

1 Introduction

In recent years, neural networks have emerged as powerful tools in scientific computing, offering new ways to approximate solutions to problems traditionally addressed by numerical methods. While classical techniques such as finite difference, finite element, and Runge-Kutta methods remain foundational for solving differential equations, neural networks provide a mesh-free, data-driven alternative that may generalise better in certain contexts.

This report investigates the viability of neural networks for approximating solutions to differential equations. Rather than comparing these methods directly with classical techniques, we focus on evaluating their performance across a range of problem types, and on identifying how their effectiveness depends on architectural choices. Specifically, we consider their application to ordinary differential equations (ODEs), including initial value problems (IVPs) and boundary value problems (BVPs). We examine their ability to interpolate and extrapolate various types of solutions. As an extension, we also investigate the use of neural networks in solving selected partial differential equations (PDEs) using similar criteria. In each case, we analyse how performance varies with different network architectures and activation functions.

This report has the following structure. Section 2 introduces the core concepts of neural networks, including their architecture and training via backpropagation. It then outlines the specifics of using neural networks to solve differential equations, illustrating the methodology with a simple example. Section 3 explores the use of neural networks for solving ODEs, dividing the discussion into IVPs and BVPs. For each, we examine a range of representative problems and analyse networks' ability to capture the underlying solution behaviour, and how that ability varies with architecture and activation function choices. Section 4 extends the investigation to PDEs, following a similar approach. We test networks on a simple problem and assess how well they satisfy the relevant constraints. Section 5 concludes the report by summarising the findings and suggesting directions for further study.

2 Preliminaries

In this section, we outline the architecture and training process of feedforward neural networks. Although various neural network architectures exist (e.g., convolutional, recurrent), this report exclusively considers feedforward networks. All references to neural networks in this report refer exclusively to this type.

Throughout, we adopt the following notation: vectors are denoted using lowercase bold symbols (e.g., $\mathbf{x}$), while matrices are written in uppercase bold (e.g., $\mathbf{W}$). We denote the output of the neural network as $\hat{y}$, which approximates a target function y.

2.1 Neural Networks Overview and Architecture

A feedforward neural network defines a function $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$, parameterised by a collection of weight matrices and bias vectors $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$. It is trained to approximate a target function $y : \mathbb{R}^n \to \mathbb{R}^m$ using observed or synthetically generated data.

The model takes an input vector $\mathbf{x} \in \mathbb{R}^n$ and propagates it forward through a sequence of L layers, each composed of individual units called *neurons*.

Each neuron in a given layer, l, performs a simple two-step operation. First, it computes a weighted sum of its inputs and adds a bias term. Then, it applies a non-linear activation function to the result. Specifically, if a neuron with weights $\mathbf{w}_i^{(l)} \in \mathbb{R}^{n_{l-1}}$ and bias $b_i^{(l)} \in \mathbb{R}$ receives an input vector $\mathbf{z}^{(l-1)} \in \mathbb{R}^{n_{l-1}}$, then its output is given by

$$z_i^{(l)} = \sigma\left((\mathbf{w}_i^{(l)})^\top \mathbf{z}^{(l-1)} + b_i^{(l)}\right),$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the neuron's fixed activation function.

A layer consists of multiple such neurons, all operating in parallel on the same input vector $\mathbf{z}^{(l-1)}$, each with its own weight vector and bias. The outputs from all neurons in the layer are collected into a vector $\mathbf{z}^{(l)} \in \mathbb{R}^{n_l}$, where n_l denotes the number of neurons in layer l. Letting $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ be the matrix whose rows are the individual neuron weight vectors $(\mathbf{w}_i^{(l)})^\top$, and $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ the vector of biases, we can express the full layer computation compactly as

$$\mathbf{z}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right),$$

with the activation function σ now applied componentwise.

The network as a whole consists of a composition of such layers. Starting from the input vector $\mathbf{z}^{(0)} = \mathbf{x}$, each successive layer transforms the output of the previous one. For a network with L layers, the computation proceeds recursively:

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}\right), \quad \text{for } l = 1, \ldots, L-1,$$

with the final output given by

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = \sigma^{(L)}(\mathbf{W}^{(L)}\mathbf{z}^{(L-1)} + \mathbf{b}^{(L)}).$$

The total set of network parameters $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$ are learned during training. The architecture is defined by the number of layers L (often referred to as the *depth* of the network), the number of neurons n_l in each layer (also referred to as the *depth*), and the choice of activation functions $\sigma^{(l)}$. The first and last layers are termed the input and output layers, respectively, while intermediate layers are termed hidden layers [1]. Figure 1 gives an illustrative diagram of a neural network architecture, with each neuron denoted by a circle, and connections indicating the output of one neuron passed as an input to neurons in the next layer.

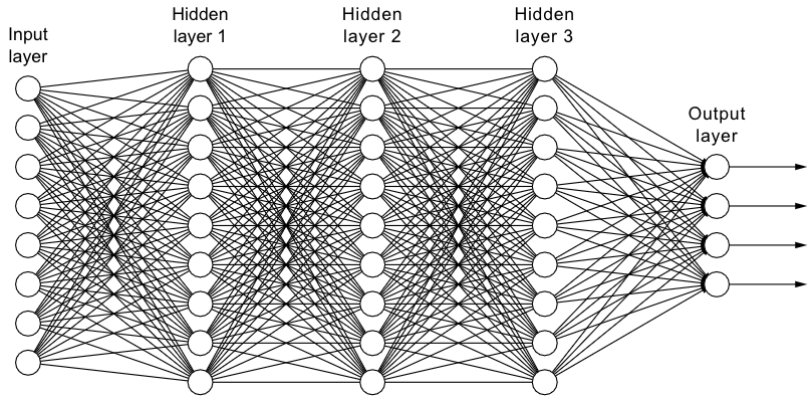


Figure 1: Illustration of a fully connected feedforward neural network with three hidden layers. Each neuron computes an affine transformation of its inputs followed by a non-linear activation.

Common activation functions used include [1]:

- **Hyperbolic tangent (tanh):** $\sigma(x) = \tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$.
  This function is smooth, bounded, and differentiable, making it particularly well-suited for approximating smooth functions.

- **Rectified Linear Unit (ReLU):** $\sigma(x) = \max(0, x)$.
  ReLU is computationally efficient and very widely used [2], but it is not differentiable at 0, and its derivative is zero for negative inputs, which can limit learning in some contexts.

- **Swish:** $\sigma(x) = x \cdot \text{sigmoid}(x) = \dfrac{x}{1 + e^{-x}}$.
  A smooth, non-monotonic function proposed as an alternative to ReLU that often improves performance in deep models [2].

In this report, we will restrict ourselves to considering these activation functions.

2.2 Training

The parameters of a neural network, namely the weight matrices and bias vectors $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$, are learned through a process called *training*. The goal of training is to find a parameter set θ^* such that the network output $\hat{y} = f_{\theta^*}(\mathbf{x})$ closely approximates the desired output y over a set of inputs $\mathbf{x} \in \mathbb{R}^n$.

This is accomplished by defining a *loss function* $\mathcal{L}(\theta)$ that quantifies the discrepancy between the network predictions and the target values across a training dataset. In this way, training becomes a minimisation problem, where the goal is to find the parameter configuration θ^* that minimises the loss function $\mathcal{L}$. When the problem is a regression-type task, a common choice is the mean squared error (MSE):

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left\| f_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right\|^2,$$

where $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$ is the training dataset of input-output pairs.

To minimise the loss function, $\mathcal{L}(\theta)$, a gradient-based optimisation method is used. This requires computing the gradient of the loss with respect to all network parameters. This is made efficient by the *backpropagation algorithm*, which systematically applies the chain rule of calculus to compute these derivatives by propagating error signals backwards through the layers of the network.

Let us denote the output of layer l as $\mathbf{z}^{(l)} \in \mathbb{R}^{n_l}$, computed via

$$\mathbf{z}^{(l)} = \sigma^{(l)}\left(\mathbf{a}^{(l)}\right), \quad \text{where} \quad \mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)},$$

and $\sigma^{(l)}$ is the activation function applied componentwise.

Define the error signal at layer l as

$$\boldsymbol{\delta}^{(l)} := \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l)}},$$

which captures the sensitivity of the loss to the pre-activation input at that layer. The error at the final layer L is computed using the derivative of the loss function with respect to the network output:

$$\boldsymbol{\delta}^{(L)} = \nabla_{\hat{\mathbf{y}}} \mathcal{L} \odot \sigma'^{(L)}\left(\mathbf{a}^{(L)}\right),$$

where $\odot$ denotes elementwise (Hadamard) product and $\sigma'^{(L)}$ is the derivative of the activation function at the final layer.

For hidden layers $l = L-1, \ldots, 1$, the errors are computed recursively using

$$\boldsymbol{\delta}^{(l)} = \left((\mathbf{W}^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)}\right) \odot \sigma'^{(l)}\left(\mathbf{a}^{(l)}\right).$$

Once the error signals are computed for each layer, the gradients of the loss with respect to the weights and biases are given by

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{z}^{(l-1)})^\top, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}.$$

These gradients are then used in an optimisation routine, such as Adam or Stochastic Gradient Descent, to update the parameters:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}, \quad \mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}},$$

where $\alpha > 0$ is a hyperparameter termed the learning rate.

This optimisation process is repeated over the training dataset in multiple passes (epochs), until convergence to a local minimum of the loss function.

This iterative update process constitutes the core of neural network training, and naturally divides the procedure into a sequence of *forward passes*, where predictions are computed by modifying and propagating forward the inputs, and *backward passes*, where gradients are propagated backwards and parameters are updated. Each training epoch can be visualised as a forward sweep through the network architecture, illustrated by Figure 1, that generates outputs, followed by a backwards sweep in which gradients are computed via backpropagation and used to adjust the network parameters.

2.3 Neural Networks for Differential Equations

When applying neural networks to solve differential equations, the nature of the problem differs fundamentally from standard supervised learning tasks. In conventional machine learning, a model learns from a dataset of labelled input-output pairs $\{(\mathbf{x}^{(i)}, y^{(i)})\}$, and is trained to minimise prediction error on unseen examples drawn from the same underlying distribution. In contrast, solving a differential equation involves finding a function that satisfies a differential constraint and associated boundary or initial conditions. No explicit data labels are given; instead, a loss function is constructed that penalises violations of the governing equation at selected collocation points.

Another key difference is that supervised learning often involves noisy training data due to measurement error or system variability. Neural networks in that context are trained to generalise despite this uncertainty. Differential equation problems, by contrast, are typically deterministic: the equations are known exactly, and the solution is expected to satisfy them precisely. As a result, the concepts of overfitting

and underfitting take on new meaning, referring to how well the learned function satisfies the equation and constraints, rather than its generalisation to unseen data.

Before analysing more complex problem classes, we conclude this section with a simple illustrative example. This serves to demonstrate the methodology outlined, reinforce the distinctions from standard supervised learning highlighted above, and validate our implementation method. It is also representative of the general method applied in subsequent sections when training neural networks in this report.

We consider the boundary value problem

$$\begin{aligned} y''(x) &= 2, \quad 0 < x < 1, \\ y(0) &= 1, \\ y(1) &= 1, \end{aligned} \tag{1}$$

whose exact solution is $y(x) = 1 + x(1 - x)$.

We approximate this solution using a neural network $\hat{y}(x) = f_\theta(x)$. The loss function penalises both deviations from the differential equation and violations of the boundary conditions:

$$\mathcal{L}(\theta) = \sum_{k=1}^{N} \left(\hat{y}''(x_k) - 2\right)^2 + \gamma \left(\hat{y}(0) - 1\right)^2 + \gamma \left(\hat{y}(1) - 1\right)^2, \tag{2}$$

where $\{x_k\}_{k=1}^N \subset (0, 1)$ are collocation points. $\gamma > 0$ is another hyperparameter that functions as a penalty coefficient, scaling the importance of $f_\theta(x)$ satisfying the boundary conditions relative to satisfying the governing equation.

The model architecture we use to solve this problem is illustrated in Figure 2. It consists of a fully connected feedforward neural network with two hidden layers, each containing five neurons. The activation function in the hidden layers is the hyperbolic tangent, $\tanh(x)$. No activation is used in the output layer, consistent with regression tasks involving continuous outputs [1].

The network is trained on 20 equally spaced points in $[0, 1]$, including the boundary values at $x = 0$ and $x = 1$. We set $\gamma = 1$ in the loss function (2), and optimise using the Adam algorithm with a fixed learning rate of $\alpha = 0.001$. Training proceeds for 2000 epochs.

Figure 3 shows the training diagnostics. The left panel illustrates the convergence of the loss during training. The right panel compares the network's prediction to the true solution. The neural network recovers the solution accurately across the domain, with minor deviations near the centre. By increasing the number of epochs, we would likely have further reduced these deviations.

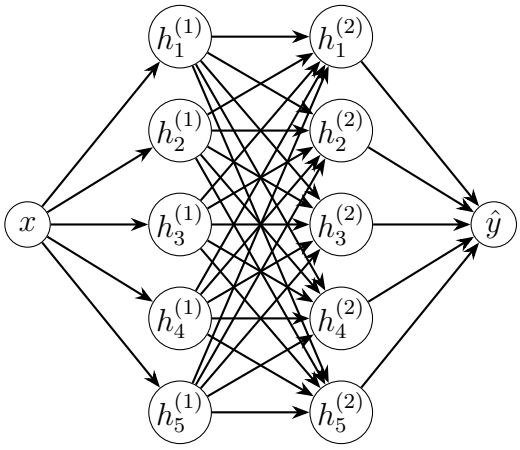


Figure 2: Fully connected feedforward neural network used to approximate the solution to (1). The network takes a scalar input x, passes it through two hidden layers with five neurons each, and outputs a scalar prediction.

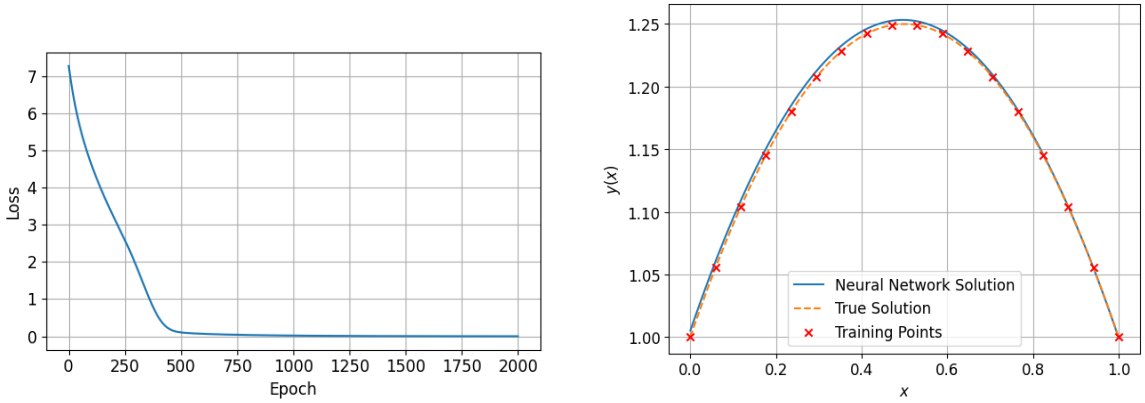
3 Ordinary Differential Equations

Having outlined the methodology for using neural networks to solve differential equations, in this section, we begin a more systematic investigation of their performance across a range of ODE problems. This section is divided into two main parts: initial value problems (IVPs) and boundary value problems (BVPs). In each case, we examine the extent to which neural networks can learn accurate solutions and how their performance is affected by architectural choices.

In the IVP setting, we consider three representative classes of problems: exponential decay, periodic solutions, and solutions containing a singularity. These problems allow us to assess both interpolation accuracy and a network's ability to extrapolate beyond the training domain.

For BVPs, our focus will remain on evaluating the quality of the approximation within the prescribed domain. Since boundary conditions are enforced at fixed endpoints, extrapolation beyond the interval is not typically meaningful. Instead, we investigate how the network adapts to constraints at both boundaries, and how well it captures internal behaviour with different choices of architecture and optimisation.

To systematically explore the influence of key design parameters, for each problem, we will vary: the number of neurons per hidden layer, the number of hidden layers and the choice of activation function. These experiments will serve to assess neural networks' ability to solve differential equation problems, along with understanding how this ability varies with architectural choices. We do not consider networks with



(a) Training loss over 2000 epochs.

(b) Neural network prediction vs true solution.

Figure 3: Training diagnostics for the neural network solution to the ODE.

different numbers of neurons in each layer, and we use the MSE loss function for all models. We choose between the tanh and ReLU activation functions, these being two of the most common activation functions chosen [1], and use the same activation functions in all networks, as is standard practice [1].

We deliberately restrict our investigation to variations in network architecture and activation functions, holding other hyperparameters fixed (such as the learning rate, optimiser type, and penalty parameters for enforcing boundary conditions). This choice simplifies the analysis and enables a clearer interpretation of the results by isolating the effects of architectural design. Our primary interest lies in the representational capability of neural networks — that is, how well different architectures can approximate solutions once trained — rather than in training efficiency. Provided that convergence is achieved, changes to hyperparameters like the learning rate or penalty weight primarily influence the speed or stability of training and not the final quality of the fit. To this end, we use the Adam optimiser, which is widely regarded as robust to hyperparameter settings such as the learning rate and penalty weight ([1], Section 8.5.4). To ensure fairness and comparability, all models were trained for the same number of epochs (typically 2500) with 100 training points in the domain, plus the boundary/initial value points. A learning rate of $\alpha = 0.001$ and penalty weight $\gamma = 100$ were used. All models were implemented in Python using the PyTorch library [3].

3.1 Initial Value Problems

3.1.1 Exponential Decay

We begin our analysis with a simple initial value problem whose solution exhibits exponential decay:

$$y'(x) = -y(x),$$
$$y(0) = 1.$$

The exact solution is $y(x) = e^{-x}$, a smooth and monotonic function defined on the entire real line. This problem provides a natural baseline for assessing the ability of neural networks to approximate well-behaved solutions and to generalise beyond the training interval.

We train a series of feedforward neural networks on this problem using the architectural variation strategy described previously. Results for both tanh and ReLU activation functions are shown in Figure 4, including error heatmaps and examples of the best and worst performing networks under each activation function.

The heatmaps in Figure 4 reveal several consistent trends. For both activation functions, increasing the number of neurons per layer leads to improved accuracy for a fixed number of layers. In contrast, increasing depth alone did not guarantee better performance, particularly when layers are narrow. The best-performing configurations are found with moderately deep networks (4–8 layers) and wider layers (10–20 neurons), though the overall sensitivity to architecture is relatively mild, likely due to the simplicity of the target function. More striking differences emerge in extrapolation. The ReLU-based networks fail to capture the exponential decay beyond the training domain, typically reverting to a linear trajectory. By contrast, the networks trained with tanh not only interpolate more accurately even in the worst case, but also exhibit qualitatively correct exponential behaviour when extrapolated to $x = 10$. Finally, we observe in our heatmap that the error for most neural network architectures using the tanh activation function has lower MSE than those using the ReLu function, regardless of architecture, suggesting that the choice of activation function is a key determinant of quality of fit.

3.1.2 Periodic Solution

We now consider an initial value problem whose solution is periodic:

$$y'(x) = \cos x,$$
$$y(0) = 0.$$

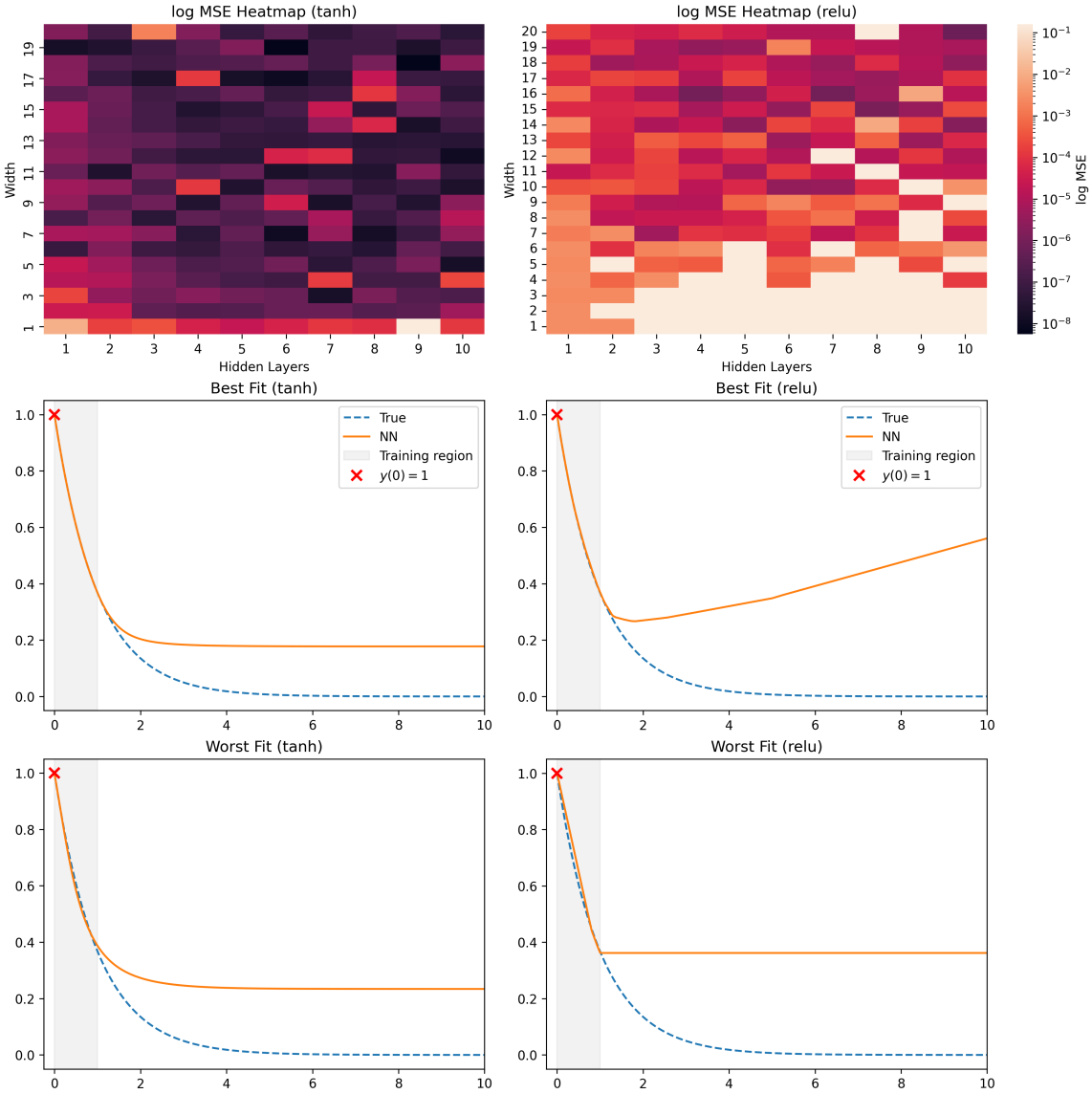


Figure 4: Comparison of architectural performance for the exponential decay problem using two activation functions. Each column shows the MSE heatmap with a log error scale, the best network fit, and the worst network fit.

The exact solution is $y(x) = \sin x$, which is smooth, bounded, and periodic with period 2π. This problem allows us to evaluate the capacity of neural networks to approximate oscillatory behaviour across a wide domain, and to generalise that behaviour beyond the domain.

We perform the same analysis as in the previous section, first analysing how the error varies for different architectures, and then examining the best solutions (we exclude the worst cases from now, as these were purely for comparison in the preceding section), shown in Figure 5.

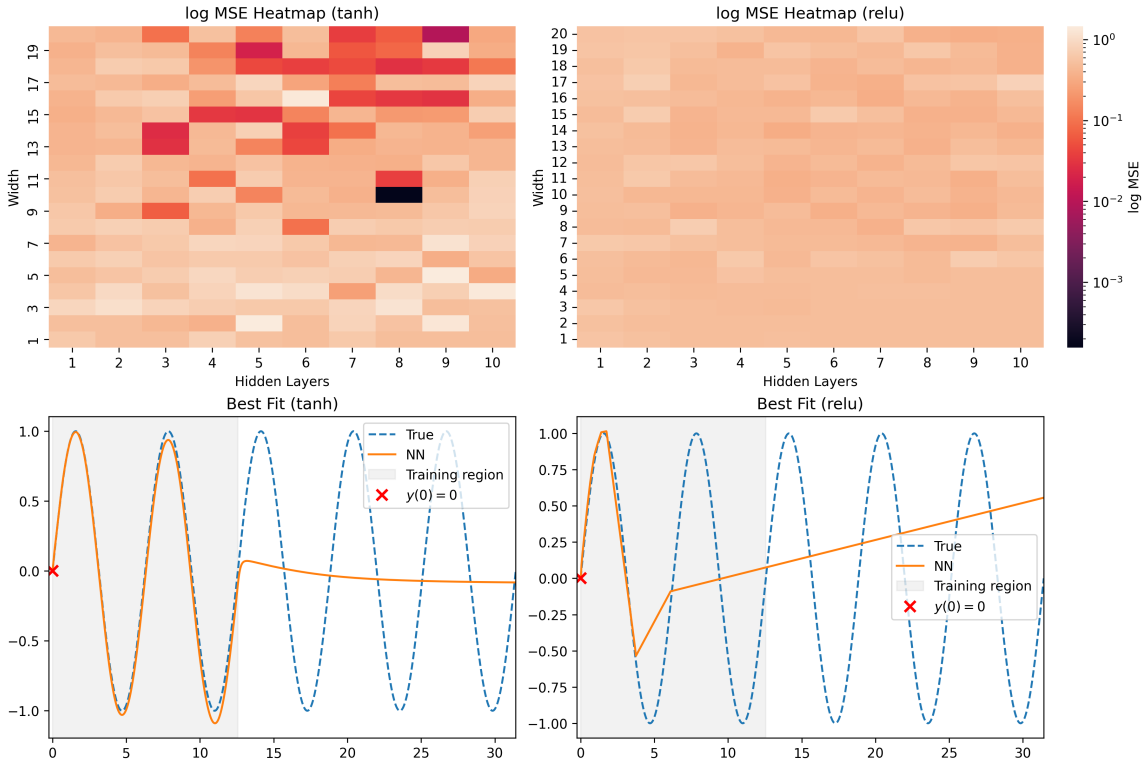


Figure 5: Comparison of architectural performance for the oscillatory problem using two activation functions. Each column shows the MSE heatmap with a log error scale, the best network fit for each activation function.

We observe that the choice of activation function has a significant impact on the network's ability to approximate this solution, the ReLU networks seeing negligible improvements in fit as the architecture was varied, and the tanh activation function yielding noticeably better fits than ReLU. This is intuitive: tanh is smooth and nonlinear, sharing important qualitative features with $\sin(x)$, while ReLU is only piecewise linear and lacks curvature. As such, we would expect ReLU networks to require substantially greater depth and width to match the expressivity of a tanh-based

network. Both activations struggle to extrapolate periodicity beyond the training domain. The predicted solutions flatten outside the interval, rather than propagating sinusoidal behaviour.

3.1.3 Singular Solution

We now turn to an initial value problem whose solution contains a singularity:

$$\begin{aligned} y'(x) &= y(x)^2, \\ y(0) &= 1, \\ y(1.05) &= -20 \end{aligned}$$

The exact solution is $y(x) = \frac{1}{1-x}$, which becomes singular at $x = 1$. This problem is useful for testing the ability of neural networks to approximate rapidly varying functions and to capture solution blow-up within a finite domain.

To mitigate instability near the singularity, we train two separate networks: one on the interval $[0, 0.95]$ using the initial condition $y(0) = 1$, and one on the interval $[1.05, 2.0]$ using the condition $y(1.05) = -20$. This approach allows us to evaluate how well neural networks can approximate the solution on either side of the singularity without numerical breakdown. Results are shown in Figure 6.

Figure 6 shows that while the ReLU activation function yields lower MSE across a broader range of architectures, the tanh activation achieves significantly better fits for select configurations. Notably, networks using tanh require at least 7 hidden layers to perform well, with accuracy dropping sharply for deeper architectures beyond 8 layers. An interesting pattern emerges with respect to width: networks with only one neuron per layer often outperform those with 2-3 neurons, suggesting that minimal width can sometimes aid in learning steep gradients near singularities. In terms of qualitative behaviour, the best tanh network closely tracks the exact solution on the right side of the singularity and extrapolates smoothly beyond the training region, converging to a nearly linear continuation, while also beginning to capture the shape on the left side. In contrast, the ReLU network produces jagged, piecewise responses outside the training domain, failing to capture the smooth decay, and completely failing to capture the structure on the left of the singularity.

3.2 Boundary Value Problems

We now consider boundary value problems (BVPs), where the solution is defined by a differential equation along with prescribed values at the boundaries of a fixed domain.

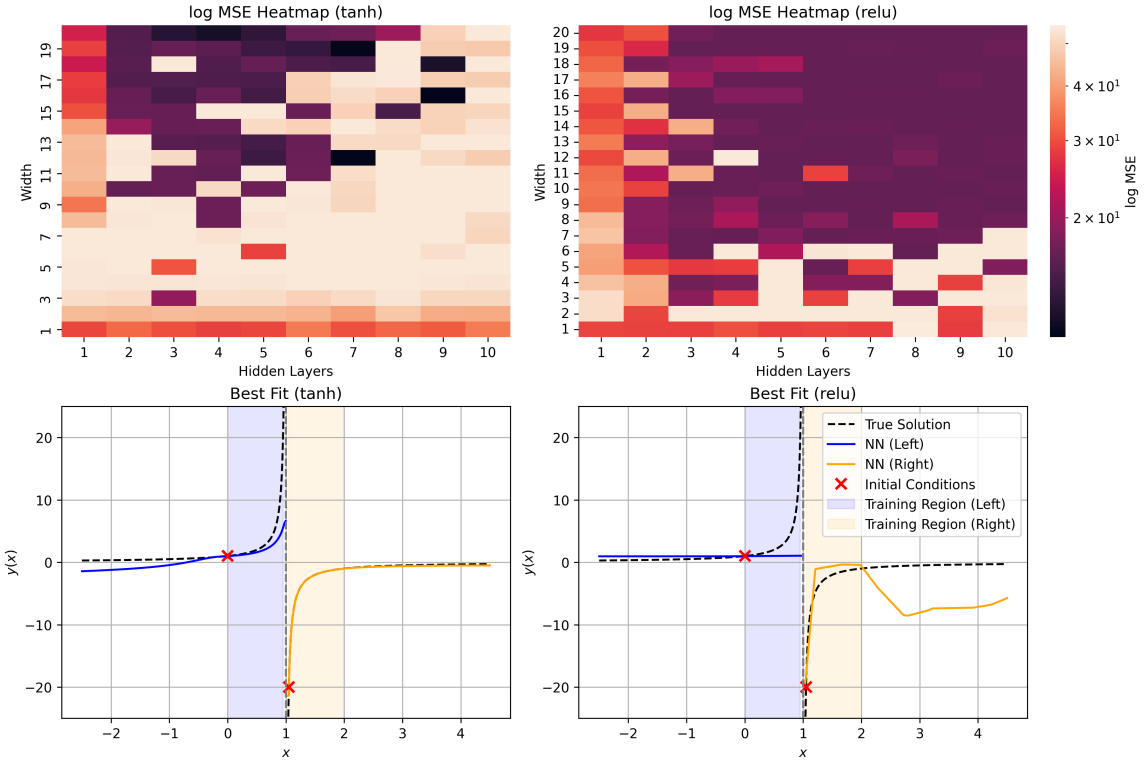


Figure 6: Comparison of architectural performance for the singular solution problem using two activation functions. Each column shows the MSE heatmap with a log error scale, the best network fit, and the worst network fit.

Unlike initial value problems, BVPs specify constraints at multiple points—typically at the endpoints of an interval—and the solution must satisfy the differential equation throughout the domain while adhering to these boundary conditions.

In this section, we investigate how well neural networks can approximate solutions to BVPs using the same methodology outlined for IVPs. However, since BVPs are defined strictly on a bounded interval, we do not consider extrapolation performance here. Instead, we focus on how accurately the networks capture the solution within the specified domain.

We consider two BVPs:

- A smooth Poisson-type problem, with known analytic solution $y(x) = sin(\pi x)$, serving as a baseline.

- A piecewise forcing problem with a discontinuous right-hand side, used to examine network behaviour under more challenging conditions.

3.2.1 Poisson Problem

We begin with a classical boundary value problem from mathematical physics:

$$\begin{aligned} -y''(x) &= \pi^2 \sin(\pi x), \quad x \in (0,1), \\ y(0) &= 0, \\ y(1) &= 0. \end{aligned}$$

This has the exact solution $y(x) = \sin(\pi x)$, which is smooth, bounded, and vanishes at both endpoints. The problem provides a simple setting to evaluate how well neural networks approximate solutions to second-order differential equations with smooth forcing and well-defined boundary conditions, and how those approximations vary with architecture.

As in the IVP analysis, we assess the effect of architectural variation on solution accuracy. For each combination of depth, width, and activation function, we compute the MSE across the domain, and visualise the results using heatmaps and the best fits, shown in Figure 7.

The ReLU activation function exhibits a clear inability to capture the curvature of the solution in this problem. Across all tested architectures, it appears to reduce error primarily by satisfying the boundary conditions, rather than accurately fitting the interior solution. In contrast, the tanh activation function consistently achieves an excellent approximation, with performance improving steadily as both depth and width are increased. This result is consistent with earlier observations, where tanh proved particularly well-suited to approximating sinusoidal solutions over the training domain.

3.2.2 Piecewise Forcing

We now consider a boundary value problem with a discontinuous right-hand side:

$$\begin{aligned} y''(x) &= \begin{cases} -1, & 0 \le x < 0.5, \\ +1, & 0.5 \le x \le 1, \end{cases} \\ y(0) &= 0, \\ y(1) &= 0. \end{aligned}$$

The exact solution is piecewise quadratic, continuous, and smooth. This example allows us to assess how well neural networks can adapt to discontinuous dynamics in the governing equation. As before, we systematically vary network architecture and

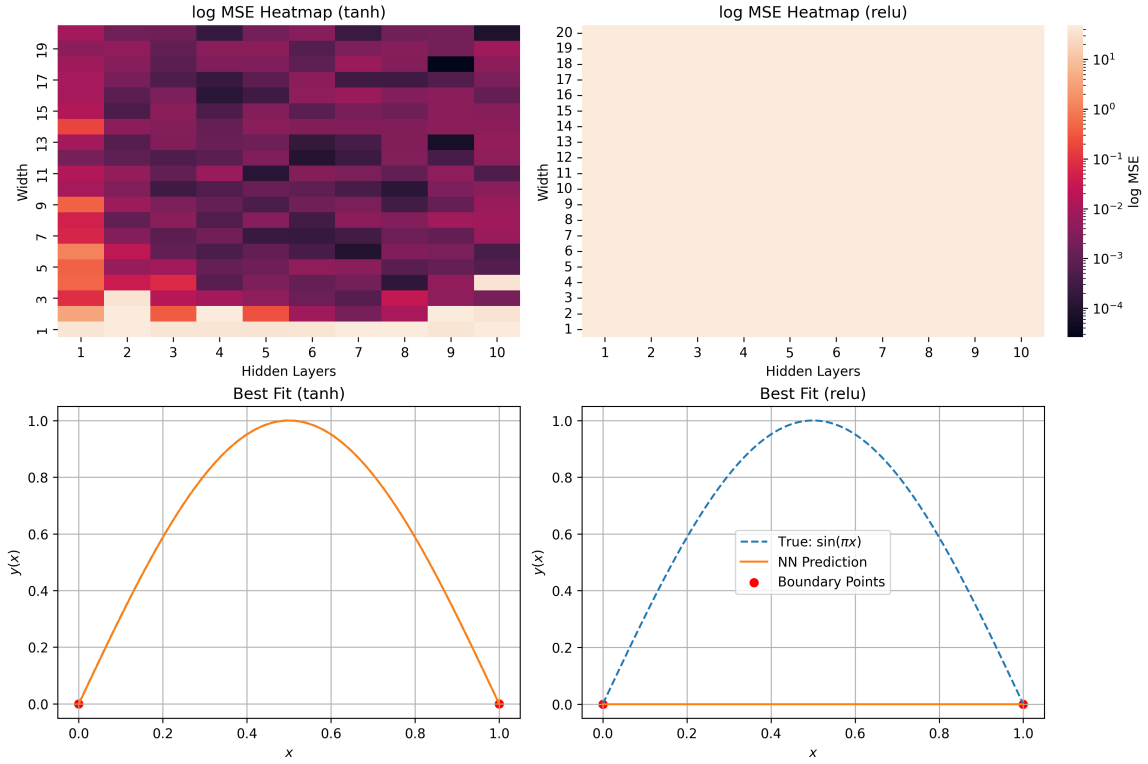


Figure 7: Comparison of architectural performance for the Poisson BVP using two activation functions. Each column shows the MSE heatmap with a log error scale, the best network fit, and the worst network fit.

activation function, compute the corresponding approximation error, and visualise representative results. The best-fitting solutions are summarised in Figure 8.

ReLU once again fails to capture the underlying structure of the solution, instead converging to a fit that satisfies only the boundary conditions, unable to capture any curvature. In contrast, networks using the tanh activation function show clear improvement with increased width and depth. It also successfully recovers the true solution extremely well.

4 Extension: Partial Differential Equations

Having assessed neural network performance across a range of ordinary differential equation problems, we now turn to a more challenging class: partial differential equations (PDEs). In this section, we consider a single representative elliptic PDE posed

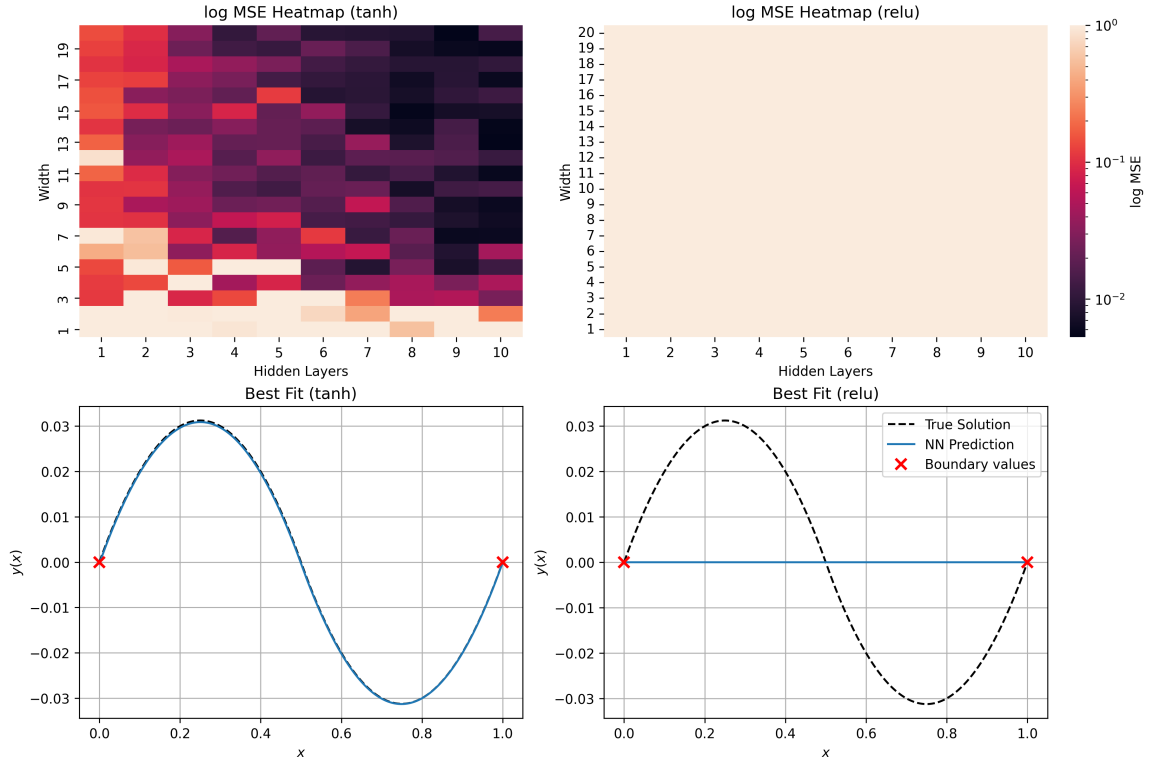


Figure 8: Comparison of architectural performance for the piecewise-forced BVP using two activation functions. Each column shows the MSE heatmap (log scale) and best-fit solution.

on a two-dimensional domain:

$$-\nabla^2 u(x,y) = f(x,y), \quad \text{for } (x,y) \in \Omega := (0,1)^2,$$
$$u(x,y) = 0, \quad \text{for } (x,y) \in \partial\Omega,$$

where the forcing function is given by

$$f(x,y) = 13\pi^2 \sin(2\pi x)\sin(3\pi y).$$

This problem admits an exact solution $u(x,y) = \sin(2\pi x)\sin(3\pi y)$, which is smooth, bounded, and vanishes on the boundary of the domain.

Our goal is to assess the ability of neural networks to recover this solution from collocation data sampled over the interior and boundary of the domain. Based on prior results, we restrict our attention to the tanh and Swish activation functions, as ReLU has been shown to perform poorly in earlier experiments. We follow a similar methodology to that used in the ODE setting: systematically varying depth and width of the network and evaluating performance using mean squared error over a

dense evaluation grid. We sample 1000 points internally and 200 points from the boundary, all points being equidistantly spaced.

In addition to architectural comparisons, we also investigate the robustness of the best-performing networks to data scarcity. Specifically, we assess how the approximation quality degrades as the number of training (collocation) points is reduced, thereby evaluating the sample efficiency of neural networks in recovering PDE solutions from limited information.

4.1 Architectural Performance on the Poisson Problem

Figure 9 shows the results of our architectural analysis. We observe consistent trends across both activation functions: increasing network width generally leads to improved accuracy, while depth plays a subtler role. The best models for both tanh and Swish achieve high-quality approximations to the true solution, closely matching the ground truth throughout most of the domain. However, the absolute error plots reveal that the Swish-based network produces a slightly more accurate fit overall, particularly along the domain boundaries. While both models perform well in the central region, the tanh network exhibits noticeable deviations near the edges, whereas the Swish model only shows small discrepancies at the corners. This suggests that the Swish activation may offer improved boundary adherence in this context, potentially due to its smooth yet non-saturating properties.

We therefore elect to use the best-performing neural network from the above found, and now examine how this network performs as training points are reduced in the next section.

4.2 Robustness to Reduced Training Data

We now assess the sensitivity of neural network approximations to the number of training points, using the best-performing architecture from the preceding analysis. We vary the number of internal and boundary training points and illustrate the resulting accuracy in Figure 10.

The heatmap shows rapid convergence to high accuracy even with relatively sparse training points. Notably, performance improvements diminish significantly beyond approximately 100 internal points and 20 boundary points, indicating a clear threshold for obtaining an accurate solution. To better illustrate this threshold effect, Figure 11 compares model predictions just below and at this threshold.

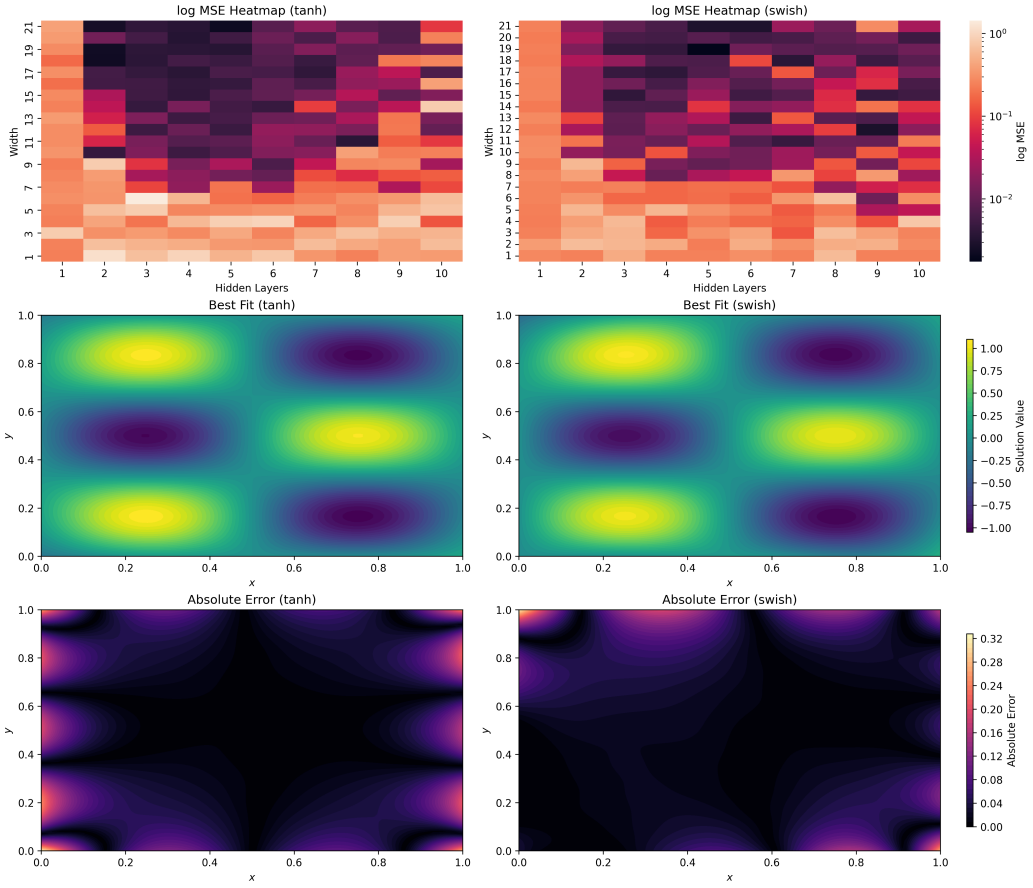


Figure 9: Comparison of neural network performance on the two-dimensional Poisson problem using the tanh and Swish activation functions. Top row: log-scaled MSE heatmaps across architectural configurations. Middle row: predicted solutions from the best-performing networks. Bottom row: corresponding absolute error plots.

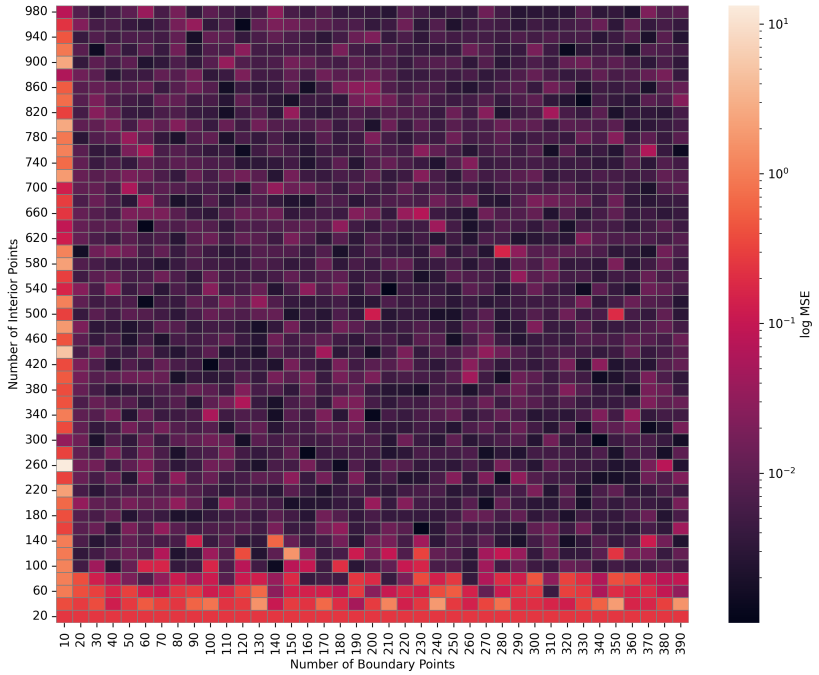


Figure 10: Mean squared error heatmap showing how approximation accuracy varies with the number of internal and boundary training points.

Figure 11 highlights how adding a modest number of training points substantially improves the accuracy and resolution of the neural network approximation. The prediction with 100 internal and 20 boundary points is notably sharper and exhibits significantly reduced error across the domain. The reported error in each case is computed as the mean squared error evaluated on 1000 equidistant points across the domain. These results confirm the robustness of neural network methods, achieving accurate PDE solutions with a relatively sparse distribution of training data.

5 Conclusion

In this report, we investigated the capability of feedforward neural networks to approximate solutions to differential equations across a variety of problem settings, focusing specifically on how performance depends on network architecture. Our analysis showed the choice of activation function significantly affected the quality of the solutions. Specifically, the ReLU activation function consistently performed poorly across most problem types, while smooth activation functions such as tanh and Swish achieved notably superior results. We also observed that neural networks generally struggled to extrapolate solutions beyond their training domains, regardless of archi-

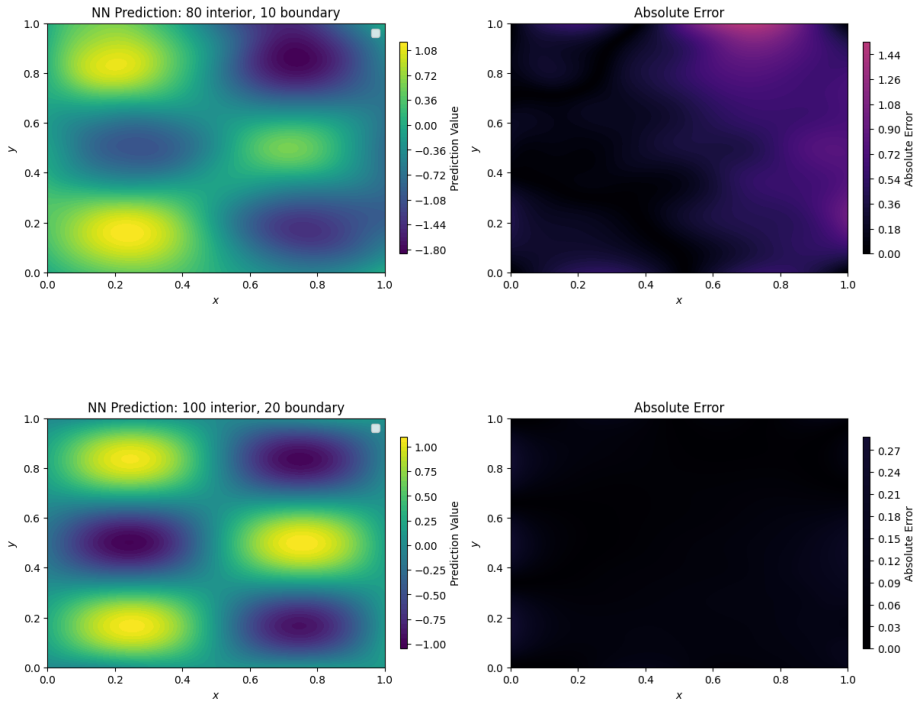


Figure 11: Comparison of neural network predictions illustrating improved accuracy with moderate increases in training data.

tectural complexity. Additionally, improvements in accuracy exhibited diminishing returns beyond certain thresholds for network depth and width. Lastly, neural networks demonstrated efficiency in solving PDEs, achieving accurate approximations even with relatively sparse training points.

For further investigation, extending our analysis to more complex PDEs, such as higher-dimensional problems, would further test the robustness and general applicability of neural network based methods. Additionally, it would be interesting to explore whether architectures with mixed activation functions lead to further improvements.

The code used to generate these results and write this report can be found at: *https://github.com/intimantripp/NeuralNetworksforPDEs*

References

[1] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[2] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[3] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).