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 generalize 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 and parameter choices. Specifically, we consider their application to ordinary differential equations (ODEs), including initial value problems (IVPs) and boundary value problems (BVPs), and examine their ability to interpolate and extrapolate various types of solutions — including exponential decay, oscillatory behaviour, and singularities. 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 hyperparameter configurations.

The remainder of this report is structured as follows:

- **Section 2** introduces the core concepts of neural networks, including their architecture and training via backpropagation, and illustrates 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 the networks' ability to capture the underlying solution behaviour.
- **Section 4** extends the investigation to PDEs, following a similar approach. We test the networks on classical problems 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 basic setup, components, and architecture of feedforward neural networks, and describe the standard approach used for training them. While multiple types of neural networks exist (e.g., convolutional neural networks, recurrent neural networks), we restrict our attention to feedforward neural 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 layer 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, the number of neurons n_l in each layer, 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. Figure 1 gives an 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.

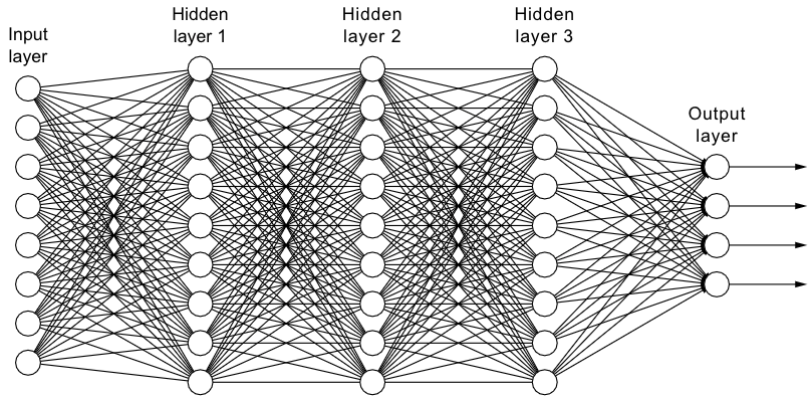


Figure 1: Illustration of a fully connected feedforward neural network with three hidden layers. Each neuron computes an affine transformation of its inputs followed by a non-linear activation.

Common activation functions used include:

- **Hyperbolic tangent (tanh):** $\sigma(x) = \tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$, often preferred for its smoothness and differentiability.
- **Rectified Linear Unit (ReLU):** $\sigma(x) = \max(0, x)$, though not ideal when higher-order derivatives are needed.

- **Leaky ReLU:** $\sigma(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases}$, for small $\alpha > 0$.
- **Sinusoidal:** $\sigma(x) = \sin(x)$, useful for representing oscillatory functions.

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 desired outputs are continuous, 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 backward 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 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, followed by a backward 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. This loss typically involves computing derivatives of the network output

with respect to its input — a task well suited to modern automatic differentiation frameworks.

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

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

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

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

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

where $\{x_k\}_{k=1}^N \subset (0, 1)$ are collocation points, and $\gamma > 0$ is a penalty coefficient.

Our model architecture 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, $\tanh(x)$, chosen for its smoothness and differentiability. No activation is used in the output layer, consistent with regression tasks involving continuous outputs.

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 (1), and optimise using the Adam algorithm with a fixed learning rate of $\alpha = 0.001$. Training proceeds for 2000 epochs.

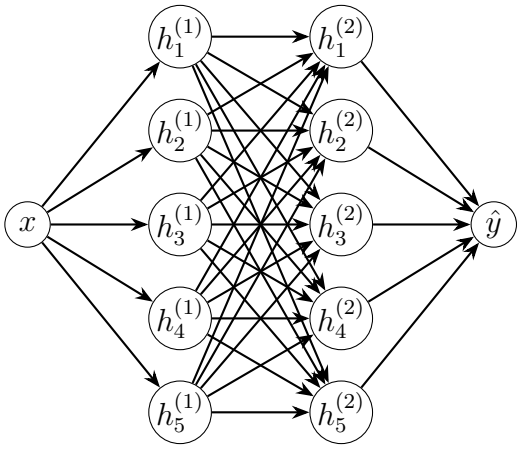
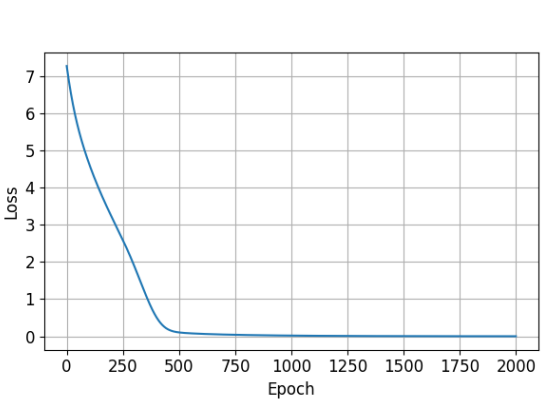
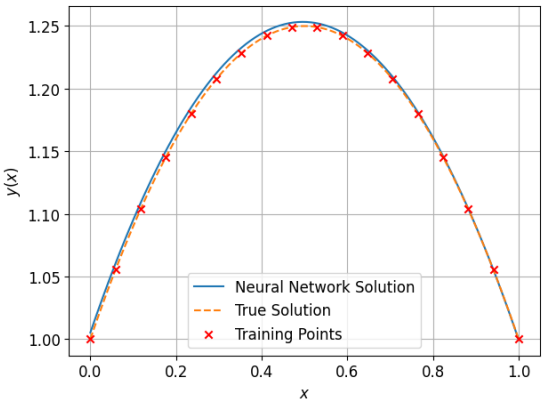


Figure 2: Fully connected feedforward neural network used to approximate the solution $\hat{y}(x)$. The network takes a scalar input x, passes it through two hidden layers with five neurons each, and outputs a scalar prediction.

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 have further reduced these deviations.



(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.

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 ordinary differential equation (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 and training choices.

In the IVP setting, we consider three representative classes of problems: exponential decay, solutions exhibiting a singularity, and periodic solutions. 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.

In both IVPs and BVPs, we systematically explore the influence of key design parameters: the number of neurons per layer, the number of hidden layers, the choice of activation function, and optimisation settings including learning rate and optimiser type. These experiments will highlight the sensitivity of neural network solvers to such hyperparameters and help identify configurations that yield stable and accurate approximations across different problem types.

3.1 Initial Value Problems

[1]

4 Extension: Partial Differential Equations

5 Conclusion

References

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