Neural Networks and Machine Learning for the resolution of Ordinary and Partial Differential Equations

Case study in Scientific Computing

Candidate Number: 1081169

1 INTRODUCTION

Neural networks' (NNs) flourishing first begun in the 1950s, focusing mainly on pattern recognition. By the 1970s it was widely thought that most of the research concerning neural networks had been exhausted, leading to some years without any significant progress. However, from the 1980s on, several societies and journals on the subject were created, guiding the subsequent, more profound research that led to the recent development of the field [1]. From within all Machine Learning (ML) techniques using neural networks, Deep Learning (DL) has had an outstanding success in a wide variety of applications [2], including pattern recognition and function approximation. However, despite all the recent progress, it is still an emerging field, whose possibilities and potential have yet to be fully studied.

Differential equations (DEs) are probably the most common type of equations that applied mathematicians deal with. Most real-world systems are described by some sort of differential equations. Many numerical methods have been developed to approximate solutions of DEs, given the difficulty of finding an analytical solution.

It is but natural to consider the usage of NNs as a potential tool to solve DEs. In the end, feed forward NNs first caught the general public's attention by their ability to approximate functions [1]. Therefore, they stand as a potentially powerful tool which provides an additional way of solving DEs. We aim to solve both linear and nonlinear differential equations, for which we will study specialised networks. In other words, the networks we will use have a loss function defined in terms of the specific equation that wants to be solved, so that each network is tailored for some DE. First, a network architecture capable of solving Ordinary Differential Equations (ODEs) will be discussed and implemented in PyTorch (Python) [3]. Subsequently, we will try to also obtain solutions for nonlinear Partial Differential Equations.

Finally, a simple network architecture with one hidden layer will be implemented in Python without using PyTorch, where the weights and biases will be updated with the Steepest Descent method and backtracking Armijo linesearch. We expect this network to have a worse performance due to its reduced complexity, and the fact that PyTorch uses C++ in the background, making it much more powerful and fast, among other reasons.

2 MODEL DESCRIPTION

2.1 DIFFERENTIAL EQUATIONS

The ODEs we will solve are second-order equations which can be written as:

$$y'' = f\left(x, y, y'\right), \qquad a < x < b, \tag{1}$$

where prime denotes differentiation with respect to x. The boundary conditions considered are either Dirichlet ($y(a) = y_a$), Von Neumann ($y'(a) = y_a$), or Robin ($y(a) + y'(a) = y_a$), likewise for $x = b$.

In contrast, the PDEs will have x and y as independent variables and are of the form:

$$\Delta u = f\left(x, y, u, u_x, u_y\right), \qquad (x, y) \in \Omega, \tag{2}$$

where Δ denotes the Laplacian of the function $u = u\left(x, y\right)$, u_x denotes differentiation with respect to x, and $\Omega := [a, b] \times [c, d]$ is the rectangular domain. The boundary conditions implemented will be the same types as for the ODE: Dirichlet ($u = g(x, y)$), Von Neumann ($u_x = g(x, y)$ or $u_y = g(x, y)$), and Robin ($u + u_x = g(x, y)$ or $u + u_y = g(x, y)$) on the boundary $\partial\Omega$. The function $g(x, y)$ will be piecewise in each of the four boundaries.

2.2 NETWORK ARCHITECTURE

In order to describe the network architecture, let L be the number of hidden layers and m the number of neurons in each layer, where $l = 1, 2, \ldots, L$. Consider also N inputs in a vector $x = (x_1, \cdots, x_N)^T \in \mathbb{R}^{N\times 1}$. The linear transformation $a\left(\cdot\right)$ from layer l can then be written as:

$$h^{(\mathrm{l})} = a\left(W^{(\mathrm{l})} h^{(\mathrm{l}-1)} + b^{(\mathrm{l})}\right), \tag{3}$$

where $h^{(0)} = x$ is the input vector, $W^{(\mathrm{l})}$ is the matrix of weights, and $b^{(\mathrm{l})}$ is the bias vector for layer l.

The output of the final hidden layer will not have the activation function applied to itself. This is because typical activation functions (ReLU, tanh, sigmoid) have a restricted range, and we want the output of our network to take whichever values solve the differential equation at hand. Hence, the output vector $\hat{y}$ may be written as:

$$\hat{y} = W^{(\mathrm{L})} h^{(\mathrm{L}-1)} + b^{(\mathrm{L})}, \tag{4}$$

where we require that $\hat{y}$ has the same number of elements as x.

2.3 LOSS FUNCTION

We will use the following residual loss function for solving ODEs:

$$\mathcal{L} = \sum_{k=2}^{N-1} \left(\hat{y}''(x_k) - f(x_k, \hat{y}(x_k), \hat{y}'(x_k))\right)^2 + \gamma(\hat{y}(x_1) - y_a)^2 + \gamma(\hat{y}(x_N) - y_b)^2, \tag{5}$$

where γ is a positive real parameter that can be chosen which weighs the effect of the boundary error on the total loss. Notice the first term gets smaller the closer the computed solution is to solving the ODE in Equation (1) for the inner points. Likewise, the boundary conditions for the outermost points are enforced with the last terms.

The PDE loss function for the inner points is defined in an analogous way:

$$\mathcal{L} = \sum_{k=2}^{N-1} \sum_{j=1}^{N-1} \left[\Delta\hat{u}(x_k, y_j) - f(x_k, y_j, \hat{u}(x_k, y_j), \hat{u}_x(x_k, y_j), \hat{u}_y(x_k, y_j))\right]^2, \tag{6}$$

where $\hat{u}$ is the network output. The boundary terms are also analogous to those in Equation (5), with the γ factor and summing over all boundary points.

2.4 PERFORMANCE METRICS

In order to select the optimal network architecture, we need to be able to compare the effect of setting a different number of layers and neurons per layer. We also need to choose an activation function, so establishing performance metrics is crucial for these decisions.

One of the metrics is, by definition, the value of the loss function. However, to test the ability of the network to capture the intricacies of each system, we will consider the loss on a validation set. The model will be trained on some points of the domain of the DE, and then validated in a slightly different set of points. As the boundaries are fixed, and hence the DE domain remains the same, we will make the set of points denser for validation, therefore passing as inputs more inner points that are new to the network. Additionally, we will also consider the training time as a measure of the performance of a network. Hence, we seek a balance between the training time and the validation loss.

A deeper network with more neurons will take more time but can capture more specific patterns or system behaviours. Therefore, the trade-off between time and validation loss depends on the application. We will perform a Pareto efficiency analysis to evaluate said trade-offs [4]. A configuration is said to be Pareto-optimal if there is no other option leading to an improvement across all performance metrics. In this case, Pareto-optimal configurations will be those such that no other configuration has a strict decrease in both validation loss and training time. Algorithm (1) details the implementation.

Algorithm 1 Pareto-optimal points

Require: Data frame *conf* with each row being a configuration and columns for L, m, training loss, validation loss, and training time. This array should have all configurations regardless of the activation used.

function IS_PARETO(costs)
  $is\_efficient \leftarrow$ 1-dimensional Boolean array of $True$ with as many elements as rows in *costs*
  n_points $\leftarrow$ number of rows (configurations) of *costs*
  for $i \leftarrow 1$ to *n_points* **do**
    if $is\_efficient[i]$ **then**
      others $\leftarrow$ *costs* excluding row i
      $others\_efficient \leftarrow is\_efficient$ excluding element i
      dominated $\leftarrow$ any(all($others[others\_efficient] \leq costs[i]$))
      $is\_efficient[i] \leftarrow$ not *dominated* ▷ If any row within the rest of points still labelled as efficient has of all its elements lower than the current one, the current one gets labelled as not efficient due to there being a state with a strict decrease in all costs
    end if
  end for
  return $is\_efficient$
end function
$costs \leftarrow conf$ (validation loss and training time columns only)
Compute Pareto points using IS_PARETO($costs$)
Plot non-Pareto and Pareto-optimal points using Matplotlib

3 PYTORCH

We will use Adam optimiser, and the learning rate used will be kept constant with a value of 10^{-3}. The training set will consist of $N < 80$ inputs evenly spread along each dimension of the domain of the DE under consideration. The maximum number of iterations will be set to 10,000. An "early stopping" mechanism is implemented which stops the training process at a lower iteration if certain conditions are met. Every 50 iterations, the current parameter values are tested on a validation set consisting of 80 points evenly spread along the domain to compute a validation loss. If, for 75 consecutive validations, the change in the validation loss is sufficiently small, the training is stopped.

Algorithm 2 Early Stopping the training process

Require: Parameters: patience = 75 (how many sufficiently small validation loss changes should be allowed before stopping the training), $\delta = 10^{-3}$ (maximum decrease in the loss between validations so that the change is sufficiently small), and the current value of the loss *current_loss*.

Initialise *best_loss* $\leftarrow \infty$
Initialise a counter $i \leftarrow 0$
Initialise Boolean variable *stop_training* $\leftarrow$ False
$\triangleright$ next lines run when a validation is done (every 50 iterations)
if *current_loss* $<$ (*best_loss* $- \delta$) **then**
    best_loss $\leftarrow$ *current_loss*
    $i \leftarrow 0$
else $\triangleright$ if the change is sufficiently small
    $i \leftarrow i + 1$
    if $i >$ patience **then**
        stop_training $\leftarrow$ True
    end if
end if

In order to choose an optimal architecture (number of hidden layers and number of neurons per layer), several of these configurations will be tested for each type of equation. Each configuration will be trained 5 different times, and the average value of the validation loss and the training time will be recorded to assess their performance. This whole process will be done with the $\tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$ and the sigmoid ($\sigma(x) = 1/(1 + e^{-x})$) activation functions. These are among the most used activations in DL, together with $\mathrm{ReLU}(x) = \max\{0, x\}$. However, we will not use ReLU due to the vanishing gradient problem when its input is 0. An additional drawback is the fact that any negative value gets set to 0. Even if the output layer does not have an activation, enforcing all hidden neuron activations to be non-negative can drastically influence the learning process for tasks as nuanced as precisely solving

DEs, which might have complex dynamics requiring significant gradient flow.

3.1 LINEAR ODE

Let us first focus on a linear, inhomogeneous ODE for $y(x)$ with Dirichlet boundary conditions (BC):

$$
\begin{aligned}
y'' &= 3y' - y + \cos(x) \qquad &\text{for } \ 0 < x < \pi, \\
y(0) &= 0, \qquad &y(\pi) = 1.
\end{aligned} \tag{7}
$$

Note there is an explicit form analytical solution against which comparisons can be made for Dirichlet BC:

$$
y\left(x\right) = \frac{e^{\frac{1}{2}(3+\sqrt{5})x} - e^{\frac{1}{2}(3-\sqrt{5})x}}{e^{\frac{1}{2}(3+\sqrt{5})\pi} - e^{\frac{1}{2}(3-\sqrt{5})\pi}} - \frac{1}{3}\sin x. \tag{8}
$$

3.1.1 PARAMETER TUNING

Let us study how the loss evolved with the number of iterations for different values of N and γ, the number of inputs and the boundary scaling factor for the loss function, respectively. From the left plot in Figure (1), we will choose $N = 40$. This is because

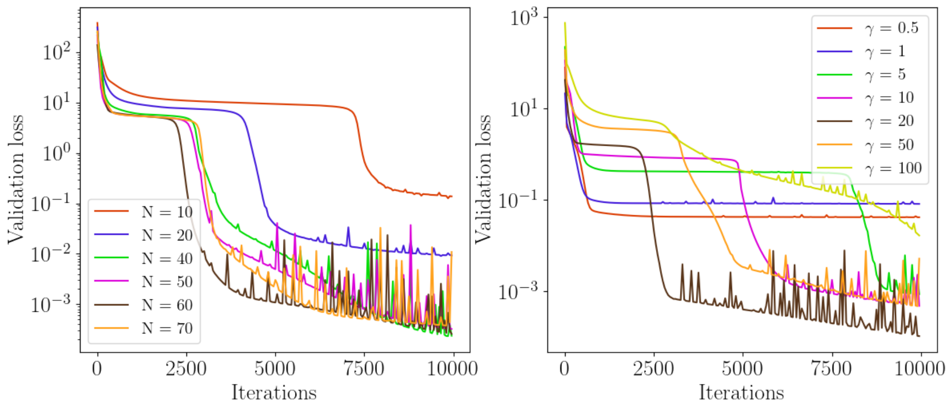

Figure 1: Validation loss against iteration number for different values of N and γ. For the left plot, $\gamma = 10$; for the right plot, $N = 40$.

it reaches a low value of the loss function not much greater than the loss achieved by larger values of N, and it oscillates with a lower amplitude, showing less variability. The value for γ will be set to 10 for the same reasons. These values will be used all throughout this section.

3.1.2 OPTIMAL ARCHITECTURE

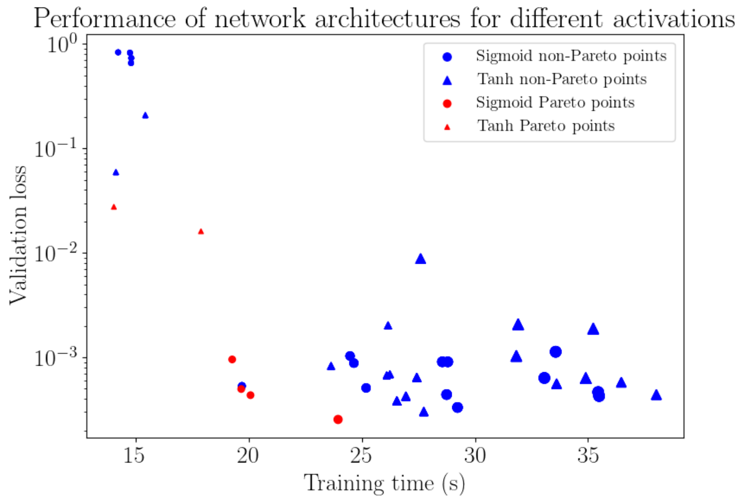

Figure 2: Performance of different network architectures under tanh and sigmoid activations, with size according to number of neurons, colour indicating passing or failing the Pareto test, shape indicating the activation function used.

The network architectures tried had $m = 10, 20, 30, 40$ and $L = 1, 2, 3, 4, 5$. For Equation (7) with the shown Dirichlet BC a general trend can be inferred from Figure (2) where simpler networks with lower neurons per layer take less training time but yield a larger validation loss, whereas more complex networks perform 1000 times better on the validation set but taking more time than the simplest ones. From the architecture configurations passing the Pareto test (marked in red) in Figure (2) there is one that takes a relatively small time near 24s, has an intermediate number of neurons, was produced with the sigmoid activation and achieved the minimum validation loss across all trials. That point corresponds to a network with $L = 3$ hidden layers and $m = 10$ neurons per hidden layer. It reached a training loss of $5.14 \cdot 10^{-5}$, a validation loss of $2.54 \cdot 10^{-4}$ in an average training time of $23.95\,\mathrm{s}$.

3.1.3 RESULTS

Figure (3) shows the solution to the ODE when the previously found optimal network architecture is applied to the ODE in Equation (7) with different mixed BC. As expected for a simple, linear ODE, a simple network with a depth of 3 layers and 10 neurons in each layer is able to perfectly capture the behaviour of the solutions for

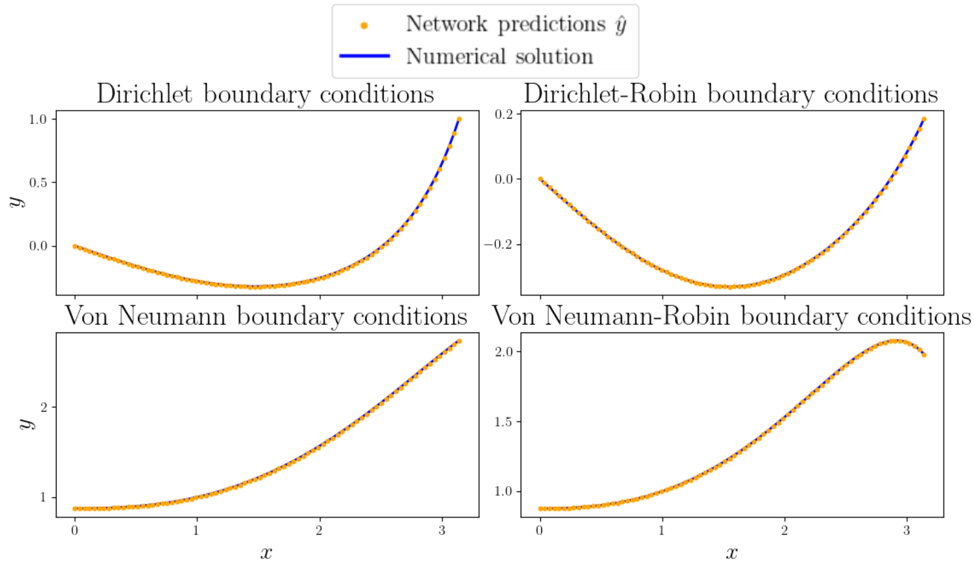

Figure 3: Analytical and predicted solutions for the ODE in Equation (7) using the Pareto-optimal configuration for different boundary conditions.

the BC displayed in Table (1). The mean validation loss across the different BC after training is $1.127 \cdot 10^{-3}$.

BC type	Left condition	Right condition	Validation loss ($\cdot 10^{-4}$)
Dirichlet	$y(0) = 0$	$y(\pi) = 1$	14.984
Dirichlet-Robin	$y(0) = 0$	$y(\pi) + y'(\pi) = 1$	20.509
Von Neumann	$y'(0) = 0$	$y'(\pi) = 1$	6.288
Von Neumann-Robin	$y'(0) = 0$	$y(\pi) + y'(\pi) = 1$	3.317

Table 1: BC and their validation loss for Figure (3).

3.2 NONLINEAR ODE

In this subsection, the same analysis as for the linear ODE will be performed. The following nonlinear ODE is proposed:

$$
\begin{aligned}
y'' &= \frac{e^{-x}\cos(yx^2)}{1+x^2} + (y')^2 y + 10\cos(6x) \qquad && 0 < x < \pi, \qquad (9)\\
y(0) &= 0, && y(\pi) = 1.
\end{aligned}
$$

3.2.1 OPTIMAL ARCHITECTURE

Due to the inherent added complexity that nonlinear ODEs present, compared to linear ODEs, we increased the range of values of L such that it ranges from 1 to 10 hidden layers. The number of neurons per layer m will still take the same values (10, 20, 30, and 40).

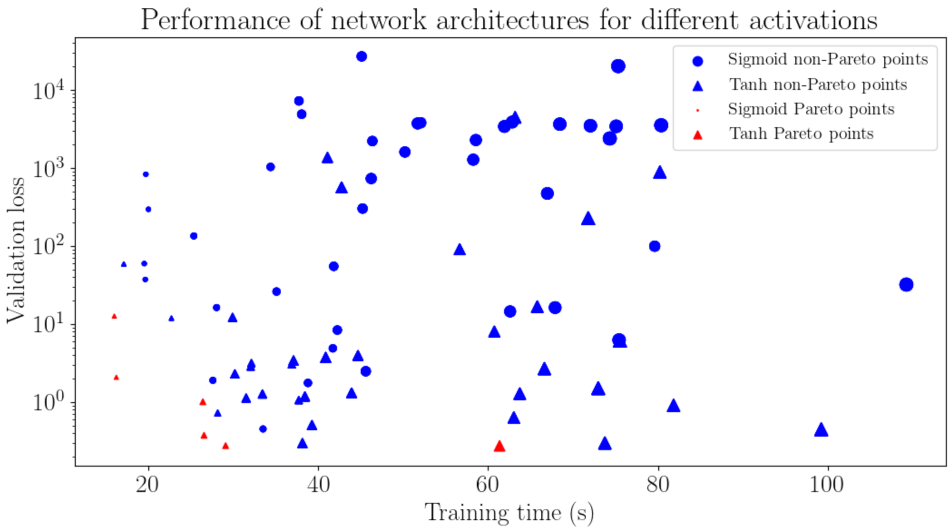

Figure 4: Performance of different network architectures under tanh and sigmoid activations, with size according to number of neurons, colour indicating passing or failing the Pareto test, shape indicating the activation function used.

A few trends can be observed from Figure (4). For this highly nonlinear ODE, the sigmoid activation was less successful in capturing the behaviour of the system than tanh: all Pareto-optimal points are achieved with the tanh activation. Tanh outperforms sigmoid because of its wider output range (-1, 1) compared to (0,1), and because it has steeper gradients near the origin. The complex interactions present in nonlinear systems require a network capable of capturing more values and gradients, which tanh does better than sigmoid.

Also, networks with a large number of hidden layers but a sparse distribution of neurons per layer (from $L = 7$ to $L = 10$, and $m = 10$) yield some of the largest values of the validation loss. This limitation is caused by the insufficient capacity of such architectures to represent the diverse range of nonlinear dynamics in the system. Conversely, architectures with 5 to 10 hidden layers and 30 or 40 neurons per layer have the ability to capture the intricate dynamics of this ODE. Hence, the breadth of the network is a crucial factor for capturing nonlinear dynamics.

Which configuration to choose as optimal within the Pareto-optimal ones depends on the specific needs of the application. For our case, computational speed is not paramount, so we will choose the architecture with the lowest validation loss: the one that took 61.36 s on average to train. Its training loss was $2.92 \cdot 10^{-3}$, its validation loss 0.273, and it has $L = 7$ hidden layers and $m = 40$ neurons per layer.

3.2.2 RESULTS

As for the linear case, we will now explore how the chosen network performs for the different boundary conditions shown in Table (1). The results are displayed in Figure (5). The validation losses obtained are displayed in Table (2).

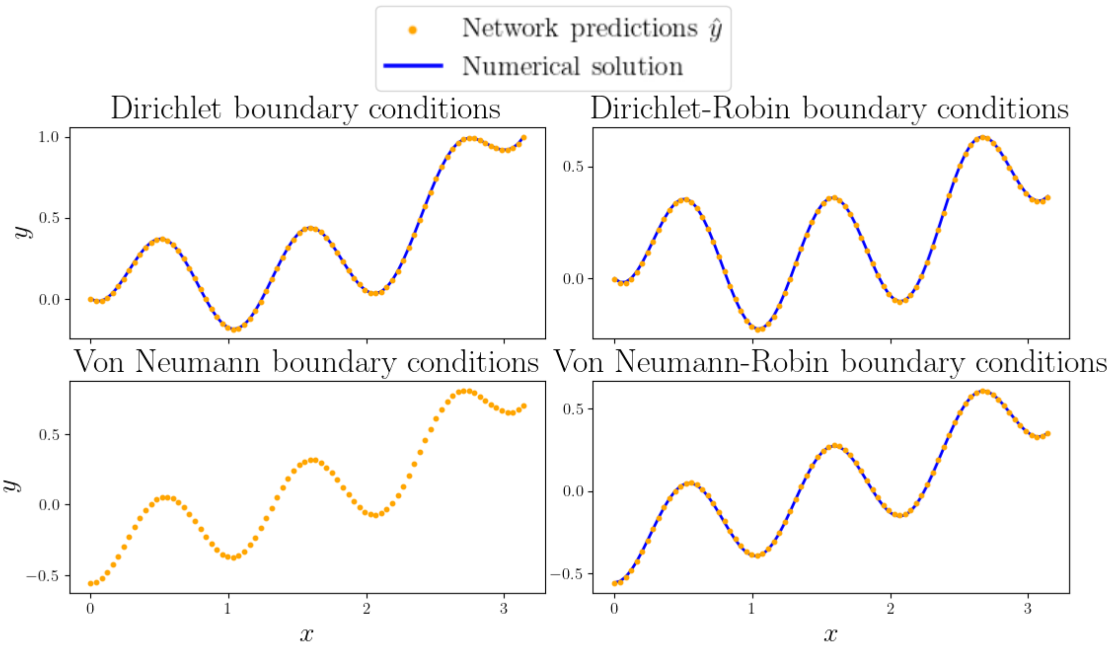

Figure 5: Numerical and predicted solutions for the ODE in Equation (9) using the Pareto-optimal configuration for different boundary conditions.

BC type	Validation loss
Dirichlet	0.602
Dirichlet-Robin	0.106
Von Neumann	0.205
Von Neumann-Robin	1.131

Table 2: Validation loss for different BC solving Equation (9).

The numerical solution in blue in Figure (5) is obtained using Scipy's "solve_bvp" by treating Equation (9) as a system of first-order ODEs. This boundary-value problem solver fails to converge when both conditions are Neumann on the bottom-left plot. This seems to be due to the inherent instability and complexity that Von Neumann conditions can add to a system. Therefore, the discretisation performed by Scipy is not able to capture the nonlinear dynamics with Neumann conditions of Equation (9). However, the NN seems to be perfectly able to approximate these dynamics, correctly satisfying the BC: $y'(0) = 0$ and $y'(\pi) = 1$.

As for the rest of BC, Figure (5) shows that the network was able to capture the nonlinear dynamics in the validation set without major problems. The mean validation loss across the different BC used was 0.511.

3.3 PDE

For this last subsection using PyTorch, we now aim to find the solution $u = u(x, y)$ of 2nd-order PDEs of two independent variables in a rectangular domain $\Omega := [0, 2\pi] \times [0, \pi]$ with BC on the boundary $\partial\Omega$. The implementation in PyTorch closely follows the ODE case, the main modification being that the network now accepts two inputs x and y instead of only one.

We will still be using $\gamma = 10$, still with Adam optimiser and a learning rate of 0.001. However, we will increase the number of inputs per axis to $N = 80$, and $N = 100$ for the validation set.

As a first step, the correctness of the implementation is checked against the known analytical solution of a PDE. When

$$\begin{aligned} \Delta u &= -\sin(x)\sin(y), \qquad && (x, y) \in (0, 2\pi) \times (0, \pi), \\ u &= 0 && (x, y) \in \partial\Omega, \end{aligned} \tag{10}$$

the solution is given by $u(x, y) = \frac{1}{2}\sin(x)\sin(y)$. Both analytical and predicted solutions on the validation set are shown in Figure (6) for a network with $L = 8$ and $m = 40$ after a training of 3,000 iterations. The validation loss was 0.843.

3.3.1 OPTIMAL ARCHITECTURE

As for the ODEs, Pareto-optimal configurations are shown in Figure (7) for $m = 20, 30, 40$, and L ranging from 5 to 10 hidden layers. Only tanh will be used now, given it outperformed the sigmoid activation for the nonlinear ODEs. As before, we

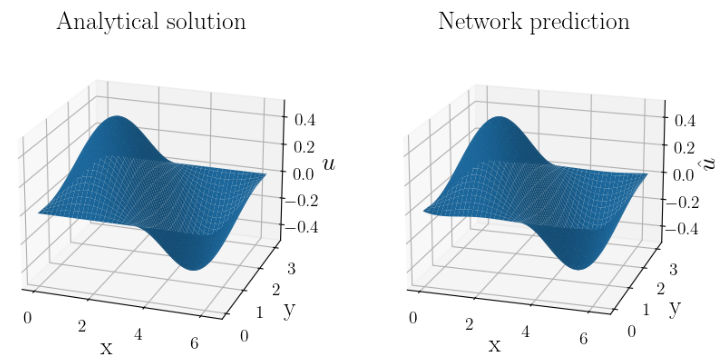

Figure 6: Analytical solution (left) and network prediction (right) for $u(x, y)$ in Equation (10).

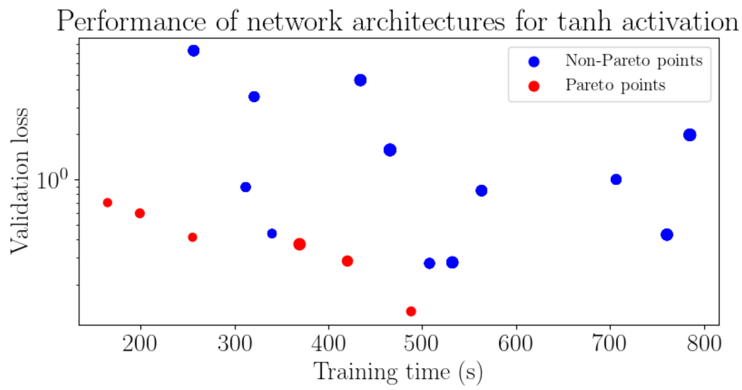

Figure 7: Performance of different network architectures, with size according to number of neurons, colour indicating passing or failing the Pareto test.

will select the network with the lowest average activation loss, which corresponds to $L = 6$ hidden layers and $m = 40$ neurons per layer. Its average training loss is 0.728, validation loss 0.132, and training time 488.17 s. Similar to the ODEs case, a certain moderate balance of layers and neurons performs better than just an increased complexity.

3.3.2 RESULTS

We will now solve the PDE on Equation (11) with different boundary conditions, the configuration chosen in the previous architecture analysis, and 5,000 iterations for the training of the network. The set of BC used is displayed in Table (3) and Figure (8) shows the results of the trained network on the validation set.

$$\Delta u = -\sin(x)\sin(y) - \cos(u), \qquad (x, y) \in (0, 2\pi) \times (0, \pi), \tag{11}$$

BC type	BC	Validation loss
Dirichlet	$u(x,0) = u(x,\pi) = u(0,y) = 0,$ $u(2\pi, y) = e^{-y}\sin(3y).$	1.615
Dirichlet-Von Neumann	$\frac{\partial u}{\partial y}(x,0) = u(x,\pi) = u(0,y) = 0,$ $u(2\pi, y) = e^{-y}\sin(3y).$	11.130
Von Neumann	$\frac{\partial u}{\partial y}(x,0) = \frac{\partial u}{\partial x}(0,y) = \frac{\partial u}{\partial x}(2\pi,y) = 0,$ $\frac{\partial u}{\partial y}(x,\pi) = -1.$	0.352
Dirichlet-Robin	$u(x,0) = u(x,\pi) + \frac{\partial u}{\partial x}(x,\pi) = \sin(3x),$ $u(0,y) = u(2\pi,y) = 0.$	14.633

Table 3: BC types and values and validation loss for the PDE in Equation (11).

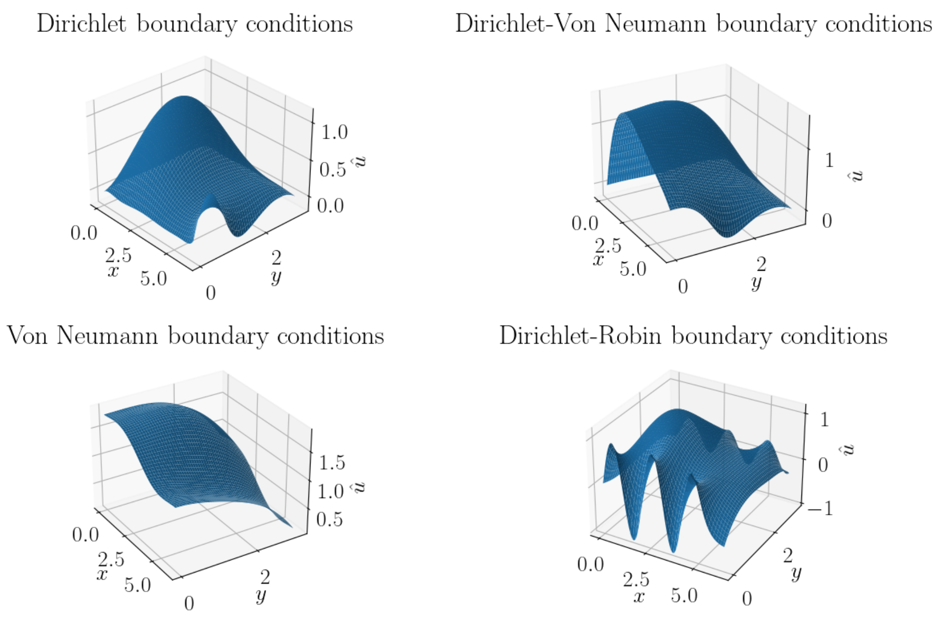

Figure 8: Predicted solutions for Equation (11) using the Pareto-optimal configuration for the BC on Table (3) after 5,000 iterations.

Note that the boundary conditions have been properly applied. The Dirichlet-Robin result accurately captured the complex system dynamics, achieving the higher loss from within the rest of the BC but still performing well. Also note how the condition $u(2\pi, y) = e^{-y}\sin(3y)$ is better captured by the network for the homogeneous

Dirichlet BC in contrast with the Von Neumann boundary condition in the top right plot of Figure (8).

4 FROM SCRATCH

In this last section, the performance of PyTorch will be compared to a full implementation of the network using the Steepest Descent (SD) method with backtracking Armijo (bArmijo) linesearch without the use of ML libraries. The network will be described by array operations which NumPy can manage [5].

Due to the difficulty of implementing backpropagation for generic architectures, only networks with one hidden layer and m neurons in it will be considered. The network is designed to solve ODEs of the form $y'' = f(x, y, y')$ over some domain, allowing for either Dirichlet or Von Neumann BC at each boundary.

4.1 NETWORK

Consider the N inputs arranged in a vector $x = (x_1, \cdots, x_N)^T$. Recall the hidden layer has m neurons. We will denote the linear transformation performed by the j^{th} neuron from the hidden layer on the input x_i as $z_{j,i}$, with weights $w_j^{(1)}$, biases $b_j^{(1)}$, and where $i = 1, \ldots, N$ and $j = 1, \ldots, m$. Also, let $a_{j,i} := \sigma(z_{j,i})$ be the output of the sigmoid activation function applied to the linear transformations from the hidden layer. The sigmoid activation was chosen due to its derivative being straightforward: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. With this notation in mind, the linear transformations occurring in neuron j for the i^{th} input can be expressed as:

$$z_{j,i} = w_j^{(1)} x_i + b_j^{(1)}. \tag{12}$$

Being a system of linear transformations, it can be expressed as a matrix product:

$$\begin{bmatrix} b_1^{(1)} & w_1^{(1)} \\ b_2^{(1)} & w_2^{(1)} \\ \vdots & \vdots \\ b_m^{(1)} & w_m^{(1)} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \end{bmatrix} = \begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & z_{1,N} \\ z_{2,1} & z_{2,2} & \cdots & z_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ z_{m,1} & z_{m,2} & \cdots & z_{m,N} \end{bmatrix}. \tag{13}$$

More compactly:

$$W^{(1)} X = Z, \tag{14}$$

where $W^{(1)} \in \mathbb{R}^{m\times 2}$, $X \in \mathbb{R}^{2\times N}$ and hence $Z \in \mathbb{R}^{m\times N}$. The sigmoid activation is then applied element-wise to Z. The notation $\sigma(Z)$ implies:

$$\sigma(Z) = \begin{bmatrix} \sigma(z_{1,1}) & \sigma(z_{1,2}) & \cdots & \sigma(z_{1,N}) \\ \sigma(z_{2,1}) & \sigma(z_{2,2}) & \cdots & \sigma(z_{2,N}) \\ \vdots & \vdots & \ddots & \vdots \\ \sigma(z_{m,1}) & \sigma(z_{m,2}) & \cdots & \sigma(z_{m,N}) \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,N} \end{bmatrix}, \tag{15}$$

or

$$A := \sigma(Z) \in \mathbb{R}^{m\times N}. \tag{16}$$

For every input, the final layer just applies a final linear transformation to the outputs of all neurons with weights $w_k^{(2)}$ and bias $b^{(2)}$:

$$\hat{y}_i = \sum_{k=1}^{m} w_k^{(2)} a_{k,i} + b^{(2)}, \tag{17}$$

which can also be expressed as a matrix operation:

$$\begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \cdots & \hat{y}_N \end{bmatrix} = \begin{bmatrix} b^{(2)} & w_1^{(2)} & w_2^{(2)} & \cdots & w_m^{(2)} \end{bmatrix} \cdot \begin{bmatrix} 1 & \cdots & 1 \\ & A & \end{bmatrix}. \tag{18}$$

More compactly:

$$Y = W^{(2)} A_{aug}, \tag{19}$$

where A has been augmented to account for the output layer's bias. Note $W^{(2)} \in \mathbb{R}^{1\times(m+1)}$, $A_{aug} \in \mathbb{R}^{(m+1)\times N}$, giving the same number of outputs as of inputs, or $Y \in \mathbb{R}^{1\times N}$.

Described like this, the implementation of the feed forward network in NumPy is straightforward.

4.2 BACKPROPAGATION

In order to perform the backpropagation for our network, we need to calculate derivatives of the loss function with respect to $w_i^{(1)}$, $w_i^{(2)}$, $b_i^{(1)}$, and $b^{(2)}$, where $i = 1, \cdots, m$. A thorough calculation can be done to obtain these analytically. Nevertheless, note that the loss function defined in Equation (5) depends on y'', the derivative of the network's output with respect to the inputs. These derivatives with respect to x will be obtained through finite differences with a stepsize ϵ. For this, slightly perturbed inputs $(x_i \pm \epsilon)$ will be needed, producing slightly perturbed outputs $(\hat{y}_{i,\pm\epsilon})$.

Finite differences will also be used to calculate derivatives of f with respect to weights and biases, given that it has a $\hat{y}_k$ and $\hat{y}'_k$ dependence. For example:

$$\begin{aligned}\frac{\partial f}{\partial w_i^{(1)}} &\approx \frac{f\left(x_k, \hat{y}(x_k; w_i^{(1)} + \epsilon), \hat{y}'(x_k; w_i^{(1)} + \epsilon)\right) - f\left(x_k, \hat{y}(x_k; w_i^{(1)}), \hat{y}'(x_k; w_i^{(1)})\right)}{\epsilon} \\ &\equiv \frac{f_{k,w_i^{(1)}+\epsilon} - f_k}{\epsilon}, \qquad (20)\end{aligned}$$

and similarly with other derivatives.

Due to the length of the calculations, we show how to get the derivatives of $\mathcal{L}$ with respect to the hidden weights $w_i^{(1)}$ and biases $b_i^{(1)}$ in the Appendix A.

4.3 STEEPEST DESCENT WITH BACKTRACKING ARMIJO

Algorithm 3 Steepest Descent with Backtracking Armijo linesearch

Input: Initial weights W_1, W_2, RHS f, tolerance tol, maximum iterations N
Output: Optimised weights W_1, W_2
Iteration count $k \leftarrow 1$
Initialise loss $loss_{\text{now}}$
while $(\|dW_1\| > \text{tol} \vee \|dW_2\| > \text{tol}) \wedge (loss_{\text{now}} > \text{tol})$ **do**
    Compute gradients dW_1, dW_2 using backpropagation
    Compute gradient norm: $norm \leftarrow \|\text{concat}(dW_1, dW_2)\|$
    Set descent directions $s_1 \leftarrow -dW_1$, $s_2 \leftarrow -dW_2$
    Random parameters $\tau, \beta \in (0, 1)$
    Initialise stepsize $a \leftarrow 1$
    Compute current loss $loss_{\text{now}}$
    Compute predicted loss $loss_{\text{next}}$ using step a
    while $loss_{\text{next}} > (loss_{\text{now}} - \beta \cdot a \cdot norm^2)$ **do**
        Reduce stepsize $a \leftarrow a \cdot \tau$
        Update $loss_{\text{now}}$ using new step a
    end while
    Update weights $W_1 \leftarrow W_1 + a \cdot s_1$
    Update weights $W_2 \leftarrow W_2 + a \cdot s_2$
    $k \leftarrow k + 1$
end while

Algorithm (3) describes the implementation of the bArmijo linesearch and SD method to minimise the loss. Notice that, unlike in the PyTorch implementation, there is not a fixed number of iterations before the training ends. Instead, we establish some minimum tolerance (10^{-1}) that either the gradient of the loss function, or the loss function itself, should achieve.

4.4 PROPOSED ODE

We will solve the following linear ODE with different BC:

$$y'' = -2x^2 + y \qquad \text{for } 0 < x < \pi. \tag{21}$$

4.5 RESULTS

As expected, the network learns at a much slower pace compared to the PyTorch networks. Apart from the fact that PyTorch runs in C++, using the SD method instead of a more developed optimiser is probably another cause for it learning slowly. Furthermore, the way we calculate the terms $\hat{y}'$ and $\hat{y}''$, which is by running the entire network for slightly perturbed inputs to calculate the finite differences, is clearly sub-optimal, as it is needed a few times per iteration. Also, to obtain the contribution to the derivatives of the loss with respect to weights and biases from $f(x, y, y')$, we again use finite differences: we run the network with slightly perturbed weights and biases. All in all, there is much that could be done to optimise this algorithm further.

All things considered, even if the algorithm works for nonlinear $f(x, y, y')$ (albeit slowly), we will only solve a linear ODE with it. The predictions for the solution of Equation (21) can be seen in Figure (9), and the exact BC are defined in Table (4).

BC type	BC	Validation loss
Dirichlet	$y(0) = 0$, $y(\pi) = 1$.	0.0163
Dirichlet -Von Neumann	$y(0) = 0$, $y'(\pi) = 1$.	20.032
Von Neumann -Dirichlet	$y'(0) = 0$, $y(\pi) = 1$.	0.394
Von Neumann	$y'(0) = 0$, $y'(2\pi) = 1$.	29.130

Table 4: BC types and values and validation loss for the ODE in Equation (21).

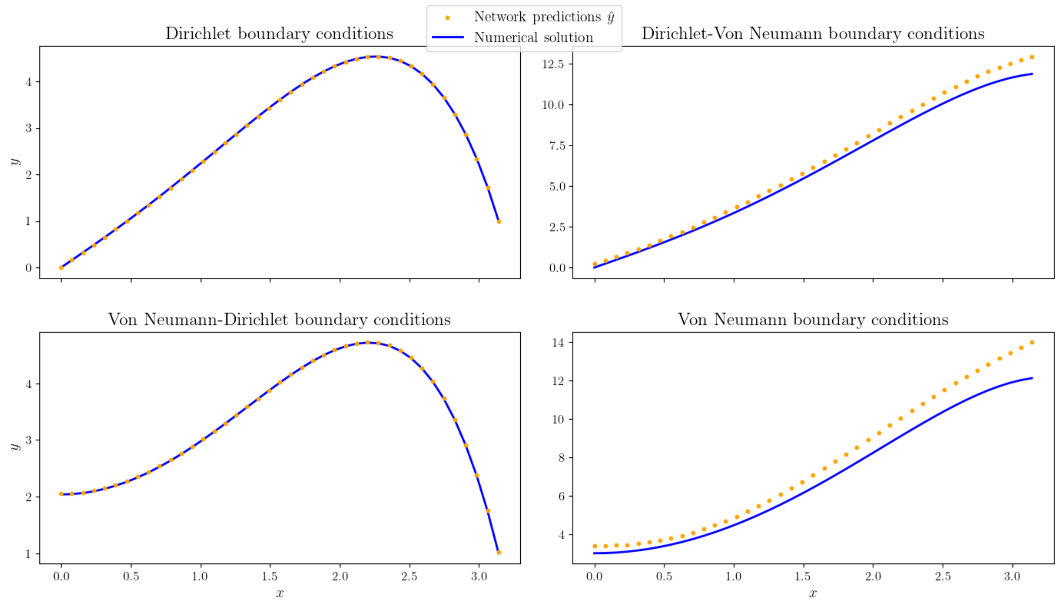

Figure 9: Numerical and predicted solutions for the ODE in Equation (21) using $m = 40$ neurons in the hidden layer for different BC.

5 DISCUSSION

We have explored the efficacy of using deep neural networks to solve both ODEs and PDEs. This paper shows the capability of neural networks with specifically designed architectures and tailored loss functions to approximate solutions to DEs instead of the traditional numerical methods.

The use of networks for solving ODEs con potentially reduce the computational complexity compared to traditional numerical methods. By identifying optimal architectures, a balance between accuracy and computational expense was achieved. For this, the Pareto efficiency analysis was useful and helped identify optimal network configurations.

The increased complexity of PDEs suggested that more complex networks should be considered. The findings suggest that deeper networks with a higher neuron count per layer were more effective at capturing the dynamics in nonlinear systems. However, not all deep networks performed well, we found there needs to be a balance between the number of layers and the number of neurons per layer. The use of the tanh activation, in particular, facilitated a better handling of the wider range of values and gradients present in nonlinear systems, as opposed to the sigmoid function, which was less effective due to its more restricted range.

The comparative analysis between PyTorch-based implementations and the one built from scratch using NumPy was useful to ratify the solid advantages of using advanced libraries like PyTorch, which optimise backpropagation in the background. This comparison not only confirmed the improved performance of library-supported implementations but also was able to solve simple ODEs in few iterations. However, as the complexity of the ODE increases, this algorithm often gets stuck or takes too long to be useful.

6 CONCLUSION

The application of neural networks in solving DEs is an intersection of ML and numerical analysis. The adaptability of neural networks can potentially outperform traditional methods in terms of both speed and accuracy, given the correct architecture and implementation. Future work could be focused on expanding the scope of the DEs addressed by these models, exploring how this process could be better generalised so that one network can solve a family of DEs, instead of only one. Developments in machine learning frameworks and computational resources are expected to improve the viability and effectiveness of neural networks in scientific computing.

References

[1] A. Yadav, M. Kumar, and N. Yadav, *An Introduction to Neural Network Methods for Differential Equations*. Springer, 2015. DOI: 10.1007/978-94-017-9816-7.

[2] L. Alzubaidi *et al.*, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 53, 2021. DOI: 10.1186/s40537-021-00444-8.

[3] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[4] D. Gerard, "Valuation equilibrium and pareto optimum," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 40, no. 7, pp. 588–592, 1959. DOI: 10.1073/pnas.40.7.588.

[5] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020. DOI: 10.1038/s41586-020-2649-2.

A BACKPROPAGATION

In this appendix we will calculate the derivative of the loss $\mathcal{L}$ as defined in Equation (5). Thanks to the linearity of differentiation, we will split the derivatives of $\mathcal{L}$ in two: the derivatives of the term for the inner points, and the derivatives of the boundary terms of the loss function.

It will be useful to express the outputs as follows:

$$\hat{y}_{i,\pm\epsilon} = \sum_{k=1}^{m} w_k^{(2)} \sigma\left(z_{k,i}\right) + b^{(2)} = \sum_{k=1}^{m} w_k^{(2)} \sigma\left(w_k^{(1)}\left(x_i \pm \epsilon\right) + b_k^{(1)}\right) + b^{(2)}. \tag{22}$$

Therefore, note the following four derivatives:

$$\frac{\partial \hat{y}_{i,\pm\epsilon}}{\partial w_j^{(1)}} = w_j^{(2)}\left(a_{j,i,\pm\epsilon}\left(1 - a_{j,i,\pm\epsilon}\right)\right)\left(x_i \pm \epsilon\right), \tag{23a}$$

$$\frac{\partial \hat{y}_{i,\pm\epsilon}}{\partial w_j^{(2)}} = a_{j,i,\pm\epsilon}, \tag{23b}$$

$$\frac{\partial \hat{y}_{i,\pm\epsilon}}{\partial b_j^{(1)}} = w_j^{(2)}\left(a_{j,i,\pm\epsilon}\left(1 - a_{j,i,\pm\epsilon}\right)\right), \tag{23c}$$

$$\frac{\partial \hat{y}_{i,\pm\epsilon}}{\partial b^{(2)}} = 1. \tag{23d}$$

A.1 BACKPROPAGATION FOR INNER LOSS $\mathcal{L}_{inner}$

The calculation of the derivatives of the inner terms of $\mathcal{L}$ with respect to $w_i^{(1)}$ can be done as shown here. We will use the notation $\hat{y}_k := \hat{y}\left(x_k\right)$. The derivatives of the inner loss are given by:

$$\begin{aligned}
\frac{\partial \mathcal{L}_{inner}}{\partial w_i^{(1)}} &= \frac{\partial}{\partial w_i^{(1)}} \sum_{k=2}^{N-1} \left(\hat{y}_k'' - f_k\right)^2 \\
&\approx 2 \sum_{k=2}^{N-1} \left(\hat{y}_k'' - f_k\right) \frac{\partial}{\partial w_i^{(1)}} \left(\frac{\hat{y}_{k,+\epsilon} - 2\hat{y}_k + \hat{y}_{k,-\epsilon}}{\epsilon^2} - f_k\right) \\
&= \frac{2}{\epsilon^2} \sum_{k=2}^{N-1} \left(\hat{y}_k'' - f_k\right) \left[\frac{\partial \hat{y}_{k,+\epsilon}}{\partial w_i^{(1)}} - 2\frac{\partial \hat{y}_k}{\partial w_i^{(1)}} + \frac{\partial \hat{y}_{k,-\epsilon}}{\partial w_i^{(1)}} - \epsilon\left(f_{k,w_i^{(1)}+\epsilon} - f_k\right)\right],
\end{aligned} \tag{24}$$

and likewise

$$\frac{\partial \mathcal{L}_{inner}}{\partial b_i^{(1)}} = \frac{2}{\epsilon^2} \sum_{k=2}^{N-1} \left(\hat{y}_k'' - f_k\right) \left[\frac{\partial \hat{y}_{k,+\epsilon}}{\partial b_i^{(1)}} - 2\frac{\partial \hat{y}_k}{\partial b_i^{(1)}} + \frac{\partial \hat{y}_{k,-\epsilon}}{\partial b_i^{(1)}} - \epsilon\left(f_{k,b_i^{(1)}+\epsilon} - f_k\right)\right]. \tag{25}$$

These can be found in terms of the $a_{i,j}$ and $a_{i,j,\pm\epsilon}$ with Equations (23a) and (23c) by doing feed forward runs of the network with $x_i \pm \epsilon$ and x_i as inputs.

In order to implement these derivatives for a backpropagation algorithm for an arbitrary number of N and m, and for the implementation to be efficient, we need a way to convert these individual derivatives into array operations that NumPy can process much faster. It is not straightforward, but by taking a close look into Equations (23a) and (23c), it can be confirmed that the following expression yields an array of the derivative of the inner loss with respect to each hidden weight and bias, $\dfrac{\partial \mathcal{L}_{inner}}{\partial W^{(1)}} \in \mathbb{R}^{m\times 2}$.

$$
\begin{aligned}
\frac{\partial \mathcal{L}_{inner}}{\partial W^{(1)}} = & \frac{2}{\epsilon^2}\left\{ \begin{bmatrix} w_1^{(2)} \\ \vdots \\ w_m^{(2)} \end{bmatrix} \odot \left(\left(A_{+\epsilon} \odot (J - A_{+\epsilon})\right) \times \left(\begin{bmatrix} \hat{y}_2'' - f_2 \\ \vdots \\ \hat{y}_{N-1}'' - f_{N-1} \end{bmatrix} \odot X_{+\epsilon} \right) \right.\right. \\
& - 2\left(A \odot (J - A)\right) \times \left(\begin{bmatrix} \hat{y}_2'' - f_2 \\ \vdots \\ \hat{y}_{N-1}'' - f_{N-1} \end{bmatrix} \odot X \right) \\
& \left. + \left(A_{-\epsilon} \odot (J - A_{-\epsilon})\right) \times \left(\begin{bmatrix} \hat{y}_2'' - f_2 \\ \vdots \\ \hat{y}_{N-1}'' - f_{N-1} \end{bmatrix} \odot X_{-\epsilon} \right) \right) \\
& + \epsilon \left(\begin{bmatrix} (f_2 - f_{2,b_1^{(1)}+\epsilon})(f_2 - f_{2,w_1^{(1)}+\epsilon}) & \cdots & (f_{N-1} - f_{N-1,b_1^{(1)}+\epsilon})(f_{N-1} - f_{N-1,w_1^{(1)}+\epsilon}) \\ \vdots & \ddots & \vdots \\ (f_2 - f_{2,b_m^{(1)}+\epsilon})(f_2 - f_{2,b_m^{(1)}+\epsilon}) & \cdots & (f_{N-1} - f_{N-1,b_m^{(1)}+\epsilon})(f_{N-1} - f_{N-1,w_m^{(1)}+\epsilon}) \end{bmatrix} \right. \\
& \left.\left. \times \begin{bmatrix} \dfrac{\hat{y}_2'' - f_2}{f_2 - f_{2,w_1^{(1)}+\epsilon}} & \dfrac{\hat{y}_2'' - f_2}{f_2 - f_{2,b_1^{(1)}+\epsilon}} \\ \vdots & \vdots \\ \dfrac{\hat{y}_{N-1}'' - f_{N-1}}{f_{N-1} - f_{N-1,w_m^{(1)}+\epsilon}} & \dfrac{\hat{y}_{N-1}'' - f_{N-1}}{f_{N-1} - f_{N-1,b_m^{(1)}+\epsilon}} \end{bmatrix} \right) \right\}, \qquad (26)
\end{aligned}
$$

where X is the input matrix as defined in Equation (13), $A_{\pm\epsilon}$ is as defined in Equation (15) but with perturbed inputs, J is an $m \times N$ matrix full of ones, $\odot$ denotes Hadamard product (element-wise with all the columns of the matrix if a vector multiplies a matrix), and $\times$ denotes matrix multiplication.

For the loss associated with the boundary points, and the derivatives of the loss with respect to the final layer's weights and biases, a similar process gives the necessary calculations.

A.2 BACKPROPAGATION FOR BOUNDARY LOSS $\mathcal{L}_b$

We will assume Dirichlet BC. Hence,

$$
\begin{aligned}
\frac{\partial \mathcal{L}_b}{\partial b_i^{(1)}} &= \gamma \frac{\partial}{\partial b_i^{(1)}} \left[(\hat{y}_1 - y_a)^2 + (\hat{y}_N - y_b)^2 \right] \\
&= 2\gamma \left((\hat{y}_1 - y_a) \frac{\partial \hat{y}_0}{\partial b_i^{(1)}} + (\hat{y}_N - y_b) \frac{\partial \hat{y_N}}{\partial b_i^{(1)}} \right) \\
&= 2\gamma \begin{bmatrix} w_1^{(2)} \\ \vdots \\ w_m^{(2)} \end{bmatrix} \odot \left\{ (\hat{y}_1 - y_a) \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \odot \left(1 - \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \right) \right. \\
&\quad \left. + (\hat{y_N} - y_b) \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \odot \left(1 - \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \right) \right\}, \qquad (27)
\end{aligned}
$$

where the column vectors are easily obtained using NumPy's slicing capabilities.

Likewise:

$$
\begin{aligned}
\frac{\partial \mathcal{L}_b}{\partial w_i^{(1)}} &= \gamma \frac{\partial}{\partial w_i^{(1)}} \left[(\hat{y}_1 - y_a)^2 + (\hat{y}_N - y_b)^2 \right] \\
&= 2\gamma \left((\hat{y}_1 - y_a) \frac{\partial \hat{y}_0}{\partial w_i^{(1)}} + (\hat{y}_N - y_b) \frac{\partial \hat{y_N}}{\partial w_i^{(1)}} \right) \\
&= 2\gamma \begin{bmatrix} w_1^{(2)} \\ \vdots \\ w_m^{(2)} \end{bmatrix} \odot \left\{ (\hat{y}_1 - y_a) \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \odot \left(1 - \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \right) x_0 \right. \\
&\quad \left. + (\hat{y_N} - y_b) \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \odot \left(1 - \begin{bmatrix} a_{1,1} \\ \vdots \\ a_{m,1} \end{bmatrix} \right) x_N \right\}, \qquad (28)
\end{aligned}
$$

The implementation in NumPy requires care with indexing, and the calculation is similar for the last layer.

B CODE

B.1 Pareto optimality

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


def is_pareto(costs):
    # Boolean array: 1 is Pareto point, 0 otherwise
    is_efficient = np.ones(costs.shape[0], dtype=bool)  # Initially, all points are Pareto
    n_points = costs.shape[0]
    # For each data point
    for i in range(n_points):
        if is_efficient[i]:
            others = np.delete(costs, i, axis=0)
            others_efficient = np.delete(is_efficient, i, axis=0)
            dominated = np.any(np.all(others[others_efficient] <= costs[i], axis=1))
            is_efficient[i] = not dominated
    return is_efficient

# Plotting settings
plt.rc('text', usetex=True)
plt.rc('font', family='arial')
plt.rcParams.update({'font.size': 20})

def linear_ode_pareto():
    # Sigmoid dataframe
    d_sigmoid = {
        'Depth': [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5],
        'Neurons per layer': [10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40],
        'Training loss': [0.67, 0.66, 0.576, 0.508, 2.55e-4, 9.71e-5, 1.18e-4, 2.2e-4, 5.14e-5, 7.61e-5, 6.77e-5, 5.34e-4, 2.17e-4, 2.34e-4, 5.76e-5, 1.22e-4, 1.27e-4, 6.54e-4, 1.2e-4, 3.69e-4],
        'Validation loss': [0.832, 0.824, 0.733, 0.657, 9.54e-4, 5.31e-4, 4.97e-4, 4.34e-4, 2.54e-4, 1.03e-3, 8.8e-4, 5.09e-4, 9.06e-4, 9.05e-4, 4.4e-4, 3.31e-4, 6.34e-4, 1.13e-3, 4.64e-4, 4.26e-4],
        'Training time': [14.23, 14.75, 14.81, 14.80, 19.27, 19.70, 19.67, 20.08, 23.95, 24.48, 24.65, 25.19, 28.56, 28.80, 28.75, 29.23, 33.07, 33.57, 35.45, 35.49]
    }
```

```python

# Tanh
d_tanh = {
    'Depth': [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5],
    'Neurons per layer': [10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40, 10, 20, 30, 40],
    'Training loss': [0.145, 1.5e-2, 3.57e-2, 7.75e-3, 1.40e-4, 2.04e-4, 7.75e-5, 1.34e-4, 7.96e-5, 1.47e-5, 1.02e-4, 2.60e-4, 2.42e-4, 3.18e-4, 5.34e-5, 4.06e-3, 5.05e-4, 3.53e-5, 2.63e-4, 4.45e-4],
    'Validation loss': [0.208, 2.76e-2, 5.92e-2, 1.61e-2, 6.87e-4, 6.71e-4, 2.02e-3, 8.26e-4, 6.41e-4, 3.82e-4, 4.24e-4, 3.02e-4, 5.57e-4, 4.39e-4, 5.76e-4, 8.81e-3, 1.88e-3, 6.32e-4, 1.03e-3, 2.07e-3],
    'Training time': [15.43, 14.04, 14.13, 17.89, 26.24, 26.10, 26.16, 23.64, 27.43, 26.55, 26.95, 27.74, 33.61, 38.02, 36.47, 27.60, 35.23, 34.90, 31.83, 31.91]
}

df_sigmoid = pd.DataFrame(data=d_sigmoid)
df_tanh = pd.DataFrame(data=d_tanh)

# Stack all configurations
df_stacked = pd.concat([df_sigmoid, df_tanh], axis=0)
df_stacked = df_stacked.reset_index(drop=True)

# Find Pareto-optimal points
pareto_points = is_pareto(df_stacked[['Training time', 'Validation loss']].values)
pareto = df_stacked[pareto_points]
pareto_indices = np.array(pareto.index.tolist())

# Create masks for non-Pareto points
mask_sigmoid = np.ones_like(df_sigmoid, dtype=bool)
mask_sigmoid[pareto_points[:20]] = False
plot_sigmoid = df_sigmoid[mask_sigmoid]

mask_tanh = np.ones_like(df_tanh, dtype=bool)
mask_tanh[pareto_points[20:]] = False
plot_tanh = df_tanh[mask_tanh]

# Plot
```

```
    fig, ax = plt.subplots(1, 1, figsize=(10, 7))
    # Plot non-Pareto sigmoid
    ax.scatter(plot_sigmoid['Training time'], plot_sigmoid['Validation loss'], s=(plot_sigmoid['Depth'] * 15), color='blue', label='Sigmoid non-Pareto points')
    # Plot non-Pareto tanh
    ax.scatter(plot_tanh['Training time'], plot_tanh['Validation loss'], s=(plot_tanh['Depth'] * 15), marker='^', color='blue', label='Tanh non-Pareto points')
    # Plot Pareto sigmoid
    ax.scatter(df_sigmoid['Training time'][pareto_points[:20]], df_sigmoid['Validation loss'][pareto_points[:20]], color='red', s=(df_sigmoid['Depth'].values[pareto_points[:20]] * 15), label='Sigmoid Pareto points')
    # Plot Pareto tanh
    ax.scatter(df_tanh['Training time'][pareto_points[20:]], df_tanh['Validation loss'][pareto_points[20:]], color='red', s=(df_tanh['Depth'].values[pareto_points[20:]] * 15), marker='^', label='Tanh Pareto points')
    ax.set_xlabel('Training time (s)')
    ax.set_ylabel('Validation loss')
    ax.set_yscale('log')
    ax.set_title('Performance of network architectures for different activations')

    plt.legend(fontsize=14, loc='upper right')
    plt.show()


def nonlinear_ode_pareto():
    # Sigmoid dataframe
    d_sigmoid = {
        'Depth': [1, 1, 1, 1,
                  2, 2, 2, 2,
                  3, 3, 3, 3,
                  4, 4, 4, 4,
                  5, 5, 5, 5,
                  6, 6, 6, 6,
                  7, 7, 7, 7,
                  8, 8, 8, 8,
                  9, 9, 9, 9,
                  10, 10, 10, 10],
        'Neurons per layer': [10, 20, 30, 40,
```

```
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40,
                               10, 20, 30, 40],
        'Training loss': [346.119, 141.127, 31.56, 51.65,
                           56.614, 0.1298, 8.14e-3, 3.25e-3,
                           281.6, 4.08e-3, 8.19e-4, 7.63e-4,
                           271.46, 2.572, 3.24e-3, 1.91e-3,
                           445.976, 6.499, 7.6e-3, 5.57e-3,
                           47.891, 5.818, 6.09e-2, 1.96e-2,
                           118.385, 68.271, 6.415, 7.62e-2,
                           102.137, 3.786, 38.646, 3.313,
                           716.180, 124.506, 7.715, 2.44e-2,
                           833.068, 103.67, 39.202, 8.735],
        'Validation loss': [828.345, 294.524, 37.12, 59.36,
                             134.200, 1.890, 16.204, 0.450,
                             1029.811, 25.974, 4.866, 1.751,
                             7225.26, 4880.43, 54.767, 8.339,
                             26901.72, 2208.715, 301.092, 2.470,
                             730.311, 3795.15, 1600.65, 99.174,
                             3716.27, 2273.974, 1275.360, 14.450,
                             3398.586, 3872.677, 470.335, 16.180,
                             3635.27, 3473.79, 3414.553, 6.217,
                             20202.64, 2383.84, 3523.38, 31.905],
        'Training time': [19.70, 20, 19.63, 19.51,
                           25.35, 27.59, 28.03, 33.50,
                           34.39, 35.09, 41.72, 38.79,
                           37.73, 38.05, 41.83, 42.25,
                           45.12, 46.38, 45.23, 45.60,
                           46.24, 52.07, 50.21, 79.63,
                           51.71, 58.57, 58.26, 62.60,
                           61.93, 62.85, 66.99, 67.90,
                           68.46, 72.06, 75.10, 75.44,
                           75.33, 74.34, 80.40, 109.28]
    }

    # Tanh
    d_tanh = {
```

```
        'Depth': [1, 1, 1, 1,
                  2, 2, 2, 2,
                  3, 3, 3, 3,
                  4, 4, 4, 4,
                  5, 5, 5, 5,
                  6, 6, 6, 6,
                  7, 7, 7, 7,
                  8, 8, 8, 8,
                  9, 9, 9, 9,
                  10, 10, 10, 10],
        'Neurons per layer': [10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40,
                              10, 20, 30, 40],
        'Training loss': [50.26, 10.204, 0.912, 9.774,
                          0.2507, 7.19e-3, 1.31e-2, 3.75e-2,
                          1.79e-2, 1.58e-2, 3.03e-3, 1.20e-2,
                          1.24e-2, 3.66e-2, 1.56e-4, 0.166,
                          5.15e-3, 0.11, 7.13e-2, 0.137,
                          0.1143, 1.38e-2, 0.679, 6.48e-3,
                          34.573, 2.28e-2, 4.39e-3, 2.92e-3,
                          129.476, 1.73e-3, 0.2113, 1.71e-2,
                          155.646, 0.3429, 1.36e-2, 1.1,
                          90.817, 4.49e-2, 0.696, 6.17e-3],
        'Validation loss': [58.58, 12.596, 2.069, 11.838,
                            1.004, 0.3731, 0.275, 0.719,
                            2.834, 3.14, 3.025, 1.053,
                            12.09, 2.285, 1.117, 1.262,
                            3.395, 1.172, 0.504, 0.296,
                            1362.28, 3.721, 1.3, 3.924,
                            90.6, 562.95, 7.981, 0.2734,
                            4445.346, 0.6278, 1.275, 16.634,
                            883.667, 0.9018, 2.671, 0.297,
                            226.5, 1.499, 0.4448, 6.11],
        'Training time': [17.1, 15.98, 16.22, 22.71,
                          26.41, 26.55, 29.09, 28.16,
                          32.05, 32.13, 36.89, 37.71,
```

```
                            29.91, 30.18, 31.51, 33.43,
                            37.09, 38.43, 39.26, 38.14,
                            41.09, 40.88, 43.93, 44.68,
                            56.66, 42.75, 60.74, 61.36,
                            63.19, 63.04, 63.74, 65.8,
                            80.25, 81.86, 66.64, 73.76,
                            71.8, 72.98, 99.25, 75.54]
    }

    df_sigmoid = pd.DataFrame(data=d_sigmoid)
    df_tanh = pd.DataFrame(data=d_tanh)

    df_stacked = pd.concat([df_sigmoid, df_tanh], axis=0)

    df_stacked = df_stacked.reset_index(drop=True)

    pareto_points = is_pareto(df_stacked[['Training time', '
    Validation loss']].values)
    pareto = df_stacked[pareto_points]
    pareto_indices = np.array(pareto.index.tolist())

    mask_sigmoid = np.ones_like(df_sigmoid, dtype=bool)
    mask_sigmoid[pareto_points[:40]] = False
    plot_sigmoid = df_sigmoid[mask_sigmoid]

    mask_tanh = np.ones_like(df_tanh, dtype=bool)
    mask_tanh[pareto_points[40:]] = False
    plot_tanh = df_tanh[mask_tanh]

    fig, ax = plt.subplots(1, 1, figsize=(10,7))

    ax.scatter(plot_sigmoid['Training time'], plot_sigmoid['
    Validation loss'], s=(plot_sigmoid['Depth']*10), color='blue',
    label='Sigmoid non-Pareto points')
    ax.scatter(plot_tanh['Training time'], plot_tanh['Validation
    loss'], s=(plot_tanh['Depth']*10), marker='^', color='blue',
    label='Tanh non-Pareto points')
    ax.scatter(df_sigmoid['Training time'][pareto_points[:40]],
    df_sigmoid['Validation loss'][pareto_points[:40]], color='red', s
    =(df_sigmoid['Depth'].values[pareto_points[:40]]*10), label='
    Sigmoid Pareto points')
    ax.scatter(df_tanh['Training time'][pareto_points[40:]], df_tanh
    ['Validation loss'][pareto_points[40:]], color='red', s=(df_tanh[
```

```
        'Depth'].values[pareto_points[40:]]*10), marker='^', label='Tanh
        Pareto points')
    ax.set_xlabel('Training time (s)')
    ax.set_ylabel('Validation loss')
    ax.set_yscale('log')
    ax.set_title('Performance of network architectures for different
     activations')

    plt.legend(fontsize=14, loc='upper right')
    plt.show()

def pde_pareto():
    # Only tanh is used
    d = {
        'Depth': [5, 5, 5,
                  6, 6, 6,
                  7, 7, 7,
                  8, 8, 8,
                  9, 9, 9,
                  10, 10, 10],
        'Neurons per layer': [20, 30, 40,
                              20, 30, 40,
                              20, 30, 40,
                              20, 30, 40,
                              20, 30, 40,
                              20, 30, 40],
        'Training loss': [0.275, 0.278, 0.449,
                          0.728, 0.577, 0.398,
                          1.048, 0.163, 0.787,
                          0.1213, 0.177, 0.402,
                          0.803, 0.201, 8.056,
                          1.3415, 0.3796, 0.4123],
        'Validation loss': [0.436, 0.412, 0.701,
                            0.132, 0.890, 0.595,
                            1.00, 0.276, 3.566,
                            0.843, 0.286, 7.232,
                            0.429, 0.280, 4.598,
                            1.984, 1.574, 0.370],
        'Training time': [340.03, 255.44, 164.88,
                          488.17, 312.00, 199.24,
                          706.61, 507.83, 321.01,
                          563.25, 420.38, 256.62,
                          760.70, 532.12, 434.23,
```

```
                              785.10, 465.72, 369.39]
    }

    df = pd.DataFrame(data=d)

    pareto_points = is_pareto(df[['Training time', 'Validation loss']].values)
    pareto = df[pareto_points]
    pareto_indices = np.array(pareto.index.tolist())

    mask = np.ones_like(df, dtype=bool)
    mask[pareto_points] = False
    df_nonpareto = df[mask]

    fig, ax = plt.subplots(1, 1, figsize=(8, 5))

    ax.scatter(df_nonpareto['Training time'], df_nonpareto['Validation loss'], s=(df_nonpareto['Depth'] * 8),
                color='blue', label='Non-Pareto points')
    ax.scatter(df['Training time'][pareto_points], df['Validation loss'][pareto_points],
                color='red', s=(df['Depth'].values[pareto_points] *8), label='Pareto points')

    ax.set_xlabel('Training time (s)')
    ax.set_ylabel('Validation loss')
    ax.set_yscale('log')
    ax.set_title('Performance of network architectures for tanh activation')

    plt.legend(fontsize=14, loc='upper right')
    plt.tight_layout()
    plt.show()

pde_pareto()
```

B.2 Early Stopping mechanism

```
    class EarlyStopping:
    def __init__(self, patience=10, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
```

```python
        self.best_loss = float('inf')
        self.wait = 0
        self.stop_training = False

    def __call__(self, current_loss):
        if current_loss < self.best_loss - self.min_delta:
            self.best_loss = current_loss
            self.wait = 0
        else:
            self.wait += 1
            if (self.wait >= self.patience):
                self.stop_training = True
```

B.3 PYTORCH NETWORK FOR ODES

```python
class OdeNN(torch.nn.Module):
    def __init__(self, input_size, hidden_size, neurons, output_size):
        # Ensure PyTorch initialises all parts of the updated feedforward net
        super(OdeNN, self).__init__()
        # Create list 'layers' to hold the layers
        self.layers = nn.ModuleList()
        # Append input layer
        self.layers.append(nn.Linear(input_size, neurons))
        # Append hidden layers
        for i in range(hidden_size-1):
            self.layers.append(nn.Linear(neurons, neurons))
        # Append output layer
        self.layers.append(nn.Linear(neurons, output_size))
        # Custom weights initialisation for tanh and sigmoid
        for layer in self.layers:
            torch.nn.init.xavier_uniform_(layer.weight)
            torch.nn.init.constant_(layer.bias, 0)

    def forward(self, x):
        for layer in self.layers[:-1]:
            # Un-comment only one activation function
            #x = torch.sigmoid(layer(x))
            x = torch.tanh(layer(x))

        # No activation for output layer
```

```
        x = self.layers[-1](x)
        return x

def ODE_loss(y_hat, x, f, bc_type, y_a, y_b, gamma):
    # Compute the first derivative of y_hat with respect to x
    y_1st = torch.autograd.grad(y_hat, x, grad_outputs=torch.ones_like(y_hat), create_graph=True, retain_graph=True, allow_unused=True)[0]
    # Compute the second derivative of y_hat with respect to x
    y_2nd = torch.autograd.grad(y_1st, x, grad_outputs=torch.ones_like(y_1st), create_graph=True, retain_graph=True, allow_unused=True)[0]

    # Compute the inner loss as the mean squared error between y_2nd and f(x)
    inner_loss = torch.sum((y_2nd - f(x, y_hat, y_1st)) ** 2)

    # Compute the left boundary term loss
    if bc_type[0] == 1: # Dirichlet
        bt_a = gamma * (y_hat[0] - y_a) ** 2
    elif bc_type[0] == 2: # Neumann
        bt_a = gamma * (y_1st[0] - y_a) ** 2
    elif bc_type[0] == 3: # Robin
        bt_a = gamma * (y_hat[0] + y_1st[0] - y_a) ** 2
    # Compute the right boundary term loss
    if  bc_type[1] == 1: # Dirichlet
        bt_b = gamma * (y_hat[-1] - y_b) ** 2
    elif bc_type[1] == 2: # Neumann
        bt_b = gamma * (y_1st[-1] - y_b) ** 2
    elif bc_type[1] == 3: # Robin
        bt_b = gamma * (y_hat[-1] + y_1st[-1] - y_b) ** 2
    # Total boundary loss
    bt_loss = bt_a + bt_b

    # Total loss
    total_loss = inner_loss + bt_loss
    return total_loss


def ODE_training(net, x, x_val, loss, optimiser, iterations, f, bc, bc_type, gamma, validate_every=50, loss_vs_iterations=False):
    # Allow differentiation wrt x and validations
    x = x.detach().requires_grad_(True)
```

```
    x_val = x_val.detach().requires_grad_(True)
    # Initialise the early stopping algorithm
    early_stopping = EarlyStopping(patience=150, min_delta=0.0001)

    val_losses = []
    epochs = []
    for iteration in range(iterations):
        # ensure no residual gradient information from previous epochs and the outputs can be differentiated wrt x
        optimiser.zero_grad()
        net.train()

        # Forward pass
        y_hat = net(x)

        # Compute loss
        total_loss = loss(y_hat, x, f, bc_type, bc[0], bc[1], gamma)

        # Compute gradient of loss wrt all parameters with requires_grad=True
        total_loss.backward()
        optimiser.step()

        # Validation
        if iteration % validate_every == 0:
            net.eval() # Network in validation mode won't get updated
            # Validation output and loss
            y_hat_val = net(x_val)
            val_loss = loss(y_hat_val, x_val, f, bc_type, bc[0], bc[1], gamma)

            val_losses.append(val_loss.item())
            epochs.append(iteration)

            # Early stopping check
            early_stopping(val_loss.item())
            if early_stopping.stop_training:
                print(f'Stopping at iteration {iteration+1}')
                break
            net.train() # Network in training mode again
```

```
        if iteration % 500 == 0:
            print(f'Iteration {iteration+1}, loss: {total_loss.item()}\nValidation loss: {val_loss.item()}')
    x.requires_grad = False
    x_val.requires_grad = False
    print(f'total loss: {total_loss.item()}; val loss: {val_loss.item()}')
    if not loss_vs_iterations:
        return total_loss.item(), val_loss.item()
    else:
        return total_loss.item(), val_loss.item(), val_losses, epochs


# PLOTTING SETTINGS
plt.rc('text', usetex=True)
plt.rc('font', family='arial')
plt.rcParams.update({'font.size': 20})

## NETWORK TRAINING
def train_func(n_inputs, n_validation, xlims, L, m, BC_type, BCs, f_torch, f_np, iterations=10000, gamma=10, loss_vs_iterations=False):
    ## TRAINING SET
    x_min = xlims[0]
    x_max = xlims[1]
    x_vals = torch.linspace(x_min, x_max, n_inputs).unsqueeze(1)
    x_np = x_vals.detach().numpy().flatten()

    # VALIDATION SET
    x_validation = torch.linspace(x_min, x_max, n_validation).unsqueeze(1)
    x_validation_np = x_validation.detach().numpy().flatten()
    x_validation = x_validation.clone().detach().requires_grad_(True)

    ts = []
    losses = []
    losses_val = []

    for i in range(5):
        # Initialise network and optimiser
```

```
        net = OdeNN(1, L, m, 1)
        opt = torch.optim.Adam(net.parameters(), 1e-3)

        # Training
        start_time = time.time()
        if not loss_vs_iterations:
            loss, val_loss = ODE_training(net, x_vals, x_validation, ODE_loss, optimiser=opt, iterations=iterations, f=f_torch, bc=BCs, bc_type=BC_type, gamma=gamma, loss_vs_iterations=loss_vs_iterations)
        else:
            loss, val_loss, val_losses, iterations = ODE_training(net, x_vals, x_validation, ODE_loss, optimiser=opt, iterations=iterations, f=f_torch, bc=BCs, bc_type=BC_type, gamma=gamma, loss_vs_iterations=loss_vs_iterations)
        end_time = time.time()
        # Training time
        training_time = end_time - start_time

        # Forward pass
        y_hat = net(x_validation)

        # Performance metrics
        losses.append(loss)
        losses_val.append(val_loss)
        ts.append(training_time)
    # Return performances
    print(f'Overall performance\n\tFinal loss: {np.mean(losses)}')
    print(f'\tValidation loss: {np.mean(losses_val)}')
    print(f'\tTraining time: {np.mean(ts)}')
    ## ANALYTICAL SOL. (SCIPY)
    def sys(x, y):
        u1, u2 = y
        f_vals = f_np(x, u1, u2)
        return np.vstack((u2, f_vals))
    def bc(ya, yb):
        if BC_type[0] == 1:
            bc_a = ya[0]
        elif BC_type[0] == 2:
            bc_a = ya[1]
        elif BC_type[0] == 3:
            bc_a = ya[0] + ya[1]
```

```
        if BC_type[1] == 1:
            bc_b = yb[0]
        elif BC_type[1] == 2:
            bc_b = yb[1]
        elif BC_type[1] == 3:
            bc_b = yb[0] + yb[1]

        return np.array([bc_a - BCs[0], bc_b - BCs[1]])
    # Solve
    y = np.zeros((2, x_validation_np.size))
    sol = solve_bvp(sys, bc, x_validation_np, y)

    if not loss_vs_iterations:
        return x_validation_np, y_hat.detach().numpy(), sol
    else:
        return x_validation_np, y_hat.detach().numpy(), sol, val_losses, iterations

# Network architecture
L = 3
m = 10
n_inputs = 30
n_validation = 80

# Boundary conditions
xlims = [0, np.pi]
BC_type = [1, 1]
bcs = [0, 1]

# ODE RHS
f_torch = lambda x, y, y1st: 3 * y1st - y + torch.cos(x)
f_np = lambda x, y, y1st: 3 * y1st - y + np.cos(x)

x, y_hat, sol = train_func(n_inputs, n_validation, xlims, L, m, BC_type, bcs, f_torch, f_np)

## PLOT
fig, ax = plt.subplots(1, 1, figsize=(10,6))
ax.scatter(x, y_hat, label='Network predictions $\hat{y}$', color='orange', s=10, zorder=2)
ax.plot(x, sol.sol(x)[0], label='Numerical solution', color='blue', linewidth=2, zorder=1)
ax.set_xlabel('$x$')
```

```
    ax.set_ylabel('$y$')
    plt.legend()
    plt.show()
```

B.4 PYTORCH NETWORK FOR PDES

```
class PdeNN(torch.nn.Module):
    def __init__(self, input_size, hidden_size, neurons_per_layer, output_size):
        super(PdeNN, self).__init__()
        # Create list to hold the layers
        self.layers = nn.ModuleList()
        # Append input layer
        self.layers.append(nn.Linear(input_size, neurons_per_layer))
        # Append hidden layers (all with same number of neurons)
        for i in range(hidden_size - 1):
            self.layers.append(nn.Linear(neurons_per_layer, neurons_per_layer))
        # Append output layer
        self.layers.append(nn.Linear(neurons_per_layer, output_size))
        # Custom weights initialisation
        for layer in self.layers:
            torch.nn.init.xavier_uniform_(layer.weight)
            torch.nn.init.constant_(layer.bias, 0)

    def forward(self, x, y):
        '''
        :param x: 2-d torch.meshgrid of x points
        :param y: 2-d torch.meshgrid of y points
        :return: 2-d torch tensor of predictions
        '''
        # Create a vector of all inputs
        xy = torch.stack((x.flatten(), y.flatten()), dim=1)
        # Nonlinear activation
        for layer in self.layers[:-1]:
            # Un-comment only one activation function
            # xy = torch.sigmoid(layer(x))
            xy = torch.tanh(layer(xy))
        # No activation for output layer
        xy = self.layers[-1](xy)
```

```python

            # Return as array of original shape
            xy = xy.view(x.shape[0], x.shape[1], x.shape[2])
            return xy

    def PDE_loss(u_hat, xx, yy, f, bc_types, bcs, gamma):
        '''
        :param bc_types: array of 4 boundary types (1: Dirichlet, 2: Von Neumann wrt x, 3: Von Neumann wrt y, 4: Robin wrt x, 5: Robin wrt y) in the order: bottom, top, left, right
        :param bcs: array of 4 anonymous functions in order: bottom, top, left, right
        :return:
        '''
        # Derivatives
        u = u_hat
        u_x = torch.autograd.grad(u, xx, grad_outputs=torch.ones_like(u_hat), create_graph=True, retain_graph=True, allow_unused=True)[0]
        u_xx = torch.autograd.grad(u_x, xx, grad_outputs=torch.ones_like(u_hat), create_graph=True, retain_graph=True, allow_unused=True)[0]

        u_y = torch.autograd.grad(u, yy, grad_outputs=torch.ones_like(u_hat), create_graph=True, retain_graph=True, allow_unused=True)[0]
        u_yy = torch.autograd.grad(u_y, yy, grad_outputs=torch.ones_like(u_hat), create_graph=True, retain_graph=True, allow_unused=True)[0]

        ## BOUNDARY LOSS
        bt_loss = 0
        #Bottom boundary
        if bc_types[0] == 1:    # Dirichlet boundary
            bt_loss += ((u_hat[0, :] - bcs[0](xx[0, :], yy[0, :])) ** 2).sum()
        elif bc_types[0] == 2:  # Von Neumann boundary wrt x
            bt_loss += ((u_x[0, :] - bcs[0](xx[0, :], yy[0, :])) ** 2).sum()
        elif bc_types[0] == 3:  # Von Neumann boundary wrt y
            bt_loss += ((u_y[0, :] - bcs[0](xx[0, :], yy[0, :])) ** 2).sum()
        elif bc_types[0] == 4:  # Robin boundary wrt x
```

```
            bt_loss += ((u_hat[0, :] + u_x[0, :] - bcs[0](xx[0, :], yy[0, :])) ** 2).sum()
    elif bc_types[0] == 5:  # Robin boundary wrt y
        bt_loss += ((u_hat[0, :] + u_y[0, :] - bcs[0](xx[0, :], yy[0, :])) ** 2).sum()
    # Top boundary
    if bc_types[1] == 1:
        bt_loss += ((u_hat[-1, :] - bcs[1](xx[-1, :], yy[-1, :])) ** 2).sum()
    elif bc_types[1] == 2:
        bt_loss += ((u_x[-1, :] - bcs[1](xx[-1, :], yy[-1, :])) ** 2).sum()
    elif bc_types[1] == 3:
        bt_loss += ((u_y[-1, :] - bcs[1](xx[-1, :], yy[-1, :])) ** 2).sum()
    elif bc_types[1] == 4:
        bt_loss += ((u_hat[-1, :] + u_x[-1, :] - bcs[1](xx[-1, :], yy[-1, :])) ** 2).sum()
    elif bc_types[1] == 5:
        bt_loss += ((u_hat[-1, :] + u_y[-1, :] - bcs[1](xx[-1, :], yy[-1, :])) ** 2).sum()
    # Left boundary
    if bc_types[2] == 1:
        bt_loss += ((u_hat[:, 0] - bcs[2](xx[:, 0], yy[:, 0])) ** 2).sum()
    elif bc_types[2] == 2:
        bt_loss += ((u_x[:, 0] - bcs[2](xx[:, 0], yy[:, 0])) ** 2).sum()
    elif bc_types[2] == 3:
        bt_loss += ((u_y[:, 0] - bcs[2](xx[:, 0], yy[:, 0])) ** 2).sum()
    elif bc_types[2] == 4:
        bt_loss += ((u_hat[:, 0] + u_x[:, 0] - bcs[2](xx[:, 0], yy[:, 0])) ** 2).sum()
    elif bc_types[2] == 5:
        bt_loss += ((u_hat[:, 0] + u_y[:, 0] - bcs[2](xx[:, 0], yy[:, 0])) ** 2).sum()
    # Right boundary
    if bc_types[3] == 1:
        bt_loss += ((u_hat[:, -1] - bcs[3](xx[:, -1], yy[:, -1])) ** 2).sum()
    elif bc_types[3] == 2:
        bt_loss += ((u_x[:, -1] - bcs[3](xx[:, -1], yy[:, -1]))
```

```
** 2).sum()
    elif bc_types[3] == 3:
        bt_loss += ((u_y[:, -1] - bcs[3](xx[:, -1], yy[:, -1])) ** 2).sum()
    elif bc_types[3] == 4:
        bt_loss += ((u_hat[:, -1] + u_x[:, -1] - bcs[3](xx[:, -1], yy[:, -1])) ** 2).sum()
    elif bc_types[3] == 5:
        bt_loss += ((u_hat[:, -1] + u_y[:, -1] - bcs[3](xx[:, -1], yy[:, -1])) ** 2).sum()

    bt_loss *= gamma

    # INNER LOSS
    inner_loss = ((u_xx[1:-1, 1:-1] + u_yy[1:-1, 1:-1] - f(u_hat[1:-1, 1:-1], xx[1:-1, 1:-1], yy[1:-1, 1:-1])) ** 2).sum()

    # TOTAL LOSS
    total_loss = bt_loss + inner_loss
    return total_loss


def PDE_training(net, xx, yy, xx_val, yy_val, loss, optimiser, iterations, f, bcs, bc_type, gamma, validate_every=50, loss_vs_iterations=False):
    # Allow gradients wrt xx and yy for loss
    xx = xx.detach().requires_grad_(True)
    xx_val = xx_val.detach().requires_grad_(True)
    yy = yy.detach().requires_grad_(True)
    yy_val = yy_val.detach().requires_grad_(True)
    early_stopping = EarlyStopping(patience=150, min_delta=0.0001)

    val_losses = []
    epochs = []
    for iteration in range(iterations):
        # ensure no residual gradient information from previous epochs and the outputs can be differentiated wrt x
        optimiser.zero_grad()
        net.train()

        # Forward pass
        u_hat = net(xx, yy)
```

```
        # Compute loss
        total_loss = loss(u_hat, xx, yy, f, bc_type, bcs, gamma)

        # Compute gradient of loss wrt all parameters with requires_grad=True
        total_loss.backward()
        optimiser.step()
        #x.detach()

        if iteration % validate_every == 0: # Validation
            net.eval() # Validation mode

            u_hat_val = net(xx_val, yy_val)
            val_loss = loss(u_hat_val, xx_val, yy_val, f, bc_type, bcs, gamma)

            val_losses.append(val_loss.item())
            epochs.append(iteration)

            # Early stopping check
            early_stopping(val_loss.item())
            if early_stopping.stop_training:
                print(f'Stopping at iteration {iteration+1}')
                break
            net.train()

        if iteration % 10 == 0:
            print(f'Iteration {iteration+1}, loss: {total_loss.item()}\nValidation loss: {val_loss.item()}')
    xx.requires_grad = False
    yy.requires_grad = False
    xx_val.requires_grad = False
    yy_val.requires_grad = False
    print(f'total loss: {total_loss.item()}; val loss: {val_loss.item()}')
    if not loss_vs_iterations:
        return total_loss.item(), val_loss.item()
    else:
        return total_loss.item(), val_loss.item(), val_losses, epochs
```

```python
def train_func(n_inputs, n_validation, xlims, ylims, L, m, bc_type, bcs, f_torch, f_np, iterations=1000, gamma=10):
    x_min, x_max = xlims[0], xlims[1]
    y_min, y_max = ylims[0], ylims[1]
    ## TRAINING SET
    x_vals = torch.linspace(x_min, x_max, n_inputs)
    y_vals = torch.linspace(y_min, y_max, n_inputs)
    xx, yy = torch.meshgrid(x_vals, y_vals, indexing='xy')
    [xx, yy] = [xx.unsqueeze(2), yy.unsqueeze(2)]
    # VALIDATION SET
    x_validation_vals = torch.linspace(x_min, x_max, n_validation)
    y_validation_vals = torch.linspace(y_min, y_max, n_validation)
    xx_val, yy_val = torch.meshgrid(x_validation_vals, y_validation_vals, indexing='xy')
    [xx_val, yy_val] = [xx_val.unsqueeze(2), yy_val.unsqueeze(2)]

    # Initialise network and optimiser
    net = PdeNN(2, L, m, 1)
    opt = torch.optim.Adam(net.parameters(), 1e-3)

    # Training
    start_time = time.time()
    loss, val_loss = PDE_training(net, xx, yy, xx_val, yy_val, PDE_loss, opt, iterations, f_torch, bcs, bc_type, gamma)
    end_time = time.time()
    training_time = end_time - start_time

    # Forward pass
    with torch.no_grad():
        u_hat = net(xx_val, yy_val).squeeze(2)
    print(f'\tLoss; {loss}\n\tValidation loss: {val_loss}\n\tTraining time: {training_time}')
    return xx_val.squeeze(2).detach().numpy(), yy_val.squeeze(2).detach().numpy(), u_hat

# Solve PDE
n_inputs = 80
n_validation = 100
L = 6
m = 40
```

```

    # Domain
    xlims = [0, 2*np.pi]
    ylims = [0, np.pi]

    # Boundary conditions
    a = lambda x, y: 0
    b = lambda x, y: 1 * torch.sin(3*y) * torch.exp(-y)
    c = lambda x, y: -1
    d = lambda x, y: torch.sin(3*x)
    bc_type = [1, 4, 1, 1]
    bcs = [d, d, a, a]

    f_torch = lambda u, x, y: -torch.sin(x)*torch.sin(y) - torch.cos(u)
    f_np = lambda x, y: -np.sin(x)*np.sin(y)
    xx_val, yy_val, u_hat = train_func(n_inputs, n_validation, xlims, ylims,
                                      L, m, bc_type, bcs, f_torch, f_np, gamma=10,
                                      iterations=10000)
    fig, ax = plt.subplots(1, 1, subplot_kw={'projection': '3d'}, figsize=(10, 6))
    ax.plot_surface(xx_val, yy_val, u_hat.detach().numpy())
    ax.set_xlabel('x', fontsize=20)
    ax.set_ylabel('y', fontsize=20)
    ax.set_zlabel(r'$u$', fontsize=20)

    plt.show()
```

B.5 FROM SCRATCH CODE

```
import numpy as np


# Define nonlinear activation
def sigmoid(z): return 1 / (1 + np.exp(-z))


# Construct neural network
def forward_propagation(X, W1, W2, N, m):
    # Hidden layer
```

```
    Z1 = W1 @ X  # Linear transformation
    A = sigmoid(Z1)  # Nonlinear activation

    # Augment A
    A_aug = np.zeros((m + 1, N + 1))
    A_aug[0, :] = np.ones((1, N + 1))
    A_aug[1:, :] = A

    # Final layer
    y_hat = W2 @ A_aug  # Linear transformation
    return y_hat, A, A_aug


# Loss function
def compute_loss(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb,
     gamma):
    def second_der(X, W1, W2, N, m, stepsize):
        '''
        returns predictions and their first and second derivatives
        '''
        # Perturb inputs for the finite differences
        X_plus = np.copy(X)
        X_plus[1, :] += stepsize
        X_minus = np.copy(X)
        X_minus[1, :] -= stepsize

        # Obtain predictions for the perturbed & unperturbed inputs
        y_hat, _, _ = forward_propagation(X, W1, W2, N, m)
        y_plus, _, _ = forward_propagation(X_plus, W1, W2, N, m)
        y_minus, _, _ = forward_propagation(X_minus, W1, W2, N, m)

        # Return the derivative through finite differences
        return [y_hat, (y_hat - y_minus) / stepsize, (y_plus - 2 *
   y_hat + y_minus) / (stepsize ** 2)]

    # Call second_der() to get derivatives
    y_hat, y_1st, y_2nd = second_der(X, W1, W2, N, m, stepsize)
    f_vals = np.array([f(X[1, i], y_hat[0, i], y_1st[0, i]) for i in
     range(1, X.shape[1] - 1)])

    # Boundary terms
    if typea == 1:
        loss_a = (y_hat[0, 0] - ya) ** 2
```

```python
    elif typea == 2:
        loss_a = (y_1st[0, 0] - ya) ** 2
    else:
        loss_a = (y_hat[0, 0] + y_1st[0, 0] - ya) ** 2
    if typeb == 1:
        loss_b = (y_hat[0, -1] - yb) ** 2
    elif typeb == 2:
        loss_b = (y_1st[0, -1] - yb) ** 2
    else:
        loss_b = (y_hat[0, -1] + y_1st[0, -1] - yb) ** 2

    # Calculate the total loss
    loss = np.sum((y_2nd[0, 1:N] - f_vals) ** 2) + gamma * (loss_a + loss_b)
    return loss, y_1st, y_2nd


def back_propagation(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma):
    ## Finite differences wrt x
    # Perturb inputs
    X_plus = np.copy(X)
    X_plus[1, :] += stepsize
    X_minus = np.copy(X)
    X_minus[1, :] -= stepsize

    # Run the unperturbed inputs through the net
    y_hat, A, A_aug = forward_propagation(X, W1, W2, N, m)
    # Compute the second derivatives
    _, y_1st, y_2nd = compute_loss(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)

    # Run the perturbed inputs through the net
    _, A_plus, A_augplus = forward_propagation(X_plus, W1, W2, N, m)
    _, A_minus, A_augminus = forward_propagation(X_minus, W1, W2, N, m)

    # Compute necessary arrays
    f_vals = np.array([f(X[1, i], y_hat[0, i], y_1st[0, i]) for i in range(1, N)])
    w2_col = W2[0, 1:].T.reshape((m, 1))  # Column vector storing values of the second weights
    residual = (y_2nd[0, 1:N] - f_vals).T.reshape((N - 1, 1))  #
```

```
Residuals of inner points

    ## Compute contribution to dL/dW1 from f(x, y)
    # Biases 1
    matrix_b1 = np.zeros((m, N - 1)) + f_vals
    for j in range(m):  # For each bias
        # Perturb it
        W1_b1 = W1.copy()
        W1_b1[j, 0] += stepsize
        # Calculate y and f with the perturbation
        y_b1, _, _ = forward_propagation(X, W1_b1, W2, N, m)
        _, y_1st_b1, _ = compute_loss(X, W1_b1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
        f_b1 = np.array([f(X[1, i], y_b1[0, i], y_1st_b1[0, i]) for i in range(1, N)])
        # Obtain the necessary array
        matrix_b1[j, :] -= f_b1
    # df/db1
    f_b1 = matrix_b1 @ residual
    # Weights 1
    matrix_w1 = np.zeros((m, N - 1)) + f_vals
    for j in range(m):  # For each weight
        # Perturb it
        W1_w1 = W1.copy()
        W1_w1[j, 1] += stepsize
        # Calculate y and f with the perturbation
        y_w1, _, _ = forward_propagation(X, W1_w1, W2, N, m)
        _, y_1st_w1, _ = compute_loss(X, W1_w1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
        f_w1 = np.array([f(X[1, i], y_w1[0, i], y_1st_w1[0, i]) for i in range(1, N)])
        # Obtain the necessary array
        matrix_w1[j, :] -= f_w1
    # df/dw1
    f_w1 = matrix_w1 @ residual
    # Then df/dW1
    dfdW1 = np.hstack((f_b1, f_w1))

    ## Obtain dL/dW1
    # Inner terms
    dA_minus = (A_minus[:, 1:N] * (1 - A_minus[:, 1:N])) @ (residual * X_minus[:, 1:N].T)
    dA = (A[:, 1:N] * (1 - A[:, 1:N])) @ (residual * X[:, 1:N].T)
```

```python
    dA_plus = (A_plus[:, 1:N] * (1 - A_plus[:, 1:N])) @ (residual * X_plus[:, 1:N].T)
    dA_total = w2_col * (dA_plus - 2 * dA + dA_minus)
    dW1 = (2 / stepsize ** 2) * dA_total + (2 / stepsize) * dfdW1

    # Boundary terms
    if typea == 1:
        bt_w1_a = 2 * gamma * w2_col * ((y_hat[0, 0] - ya) * A[:, 0] * (1 - A[:, 0]) * X[1, 0]).reshape((m, 1))
        bt_b1_a = 2 * gamma * w2_col * ((y_hat[0, 0] - ya) * A[:, 0] * (1 - A[:, 0])).reshape((m, 1))
    elif typea == 2:
        bt_w1_a = ((2 * gamma / stepsize) * w2_col *
                   (y_1st[0, 0] - ya) * (A[:, 0] * (1 - A[:, 0]) * X[1, 0] - A_minus[:, 0] * (1 - A_minus[:, 0]) * (X[1, 0] - stepsize)))
        bt_b1_a = ((2 * gamma / stepsize) * w2_col *
                   (y_1st[0, 0] - ya) * (A[:, 0] * (1 - A[:, 0]) -
                                         A_minus[:, 0] * (1 - A_minus[:, 0])))

    if typeb == 1:
        bt_w1_b = 2 * gamma * w2_col * ((y_hat[0, -1] - yb) * A[:, -1] * (1 - A[:, -1]) * X[1, -1]).reshape((m, 1))
        bt_b1_b = 2 * gamma * w2_col * ((y_hat[0, -1] - yb) * A[:, -1] * (1 - A[:, -1])).reshape((m, 1))
    elif typeb == 2:
        bt_w1_b = ((2 * gamma / stepsize) * w2_col *
                   (y_1st[0, -1] - yb) * (A[:, -1] * (1 - A[:, -1]) * X[1, -1] - A_minus[:, -1] * (1 - A_minus[:, -1]) * (X[1, -1] - stepsize)))
        bt_b1_b = ((2 * gamma / stepsize) * w2_col *
                   (y_1st[0, -1] - yb) * (A[:, -1] * (1 - A[:, -1]) - A_minus[:, -1] * (1 - A_minus[:, -1])))

    bt_w1 = bt_w1_a + bt_w1_b
    bt_b1 = bt_b1_a + bt_b1_b
    # Total derivatives dL/dW1
    dW1[:, 0] += bt_b1[:, 0]
    dW1[:, 1] += bt_w1[:, 0]

    ## Compute contribution to dL/dW2 from f(x, y)
    # Bias 2
```

```
    matrix_b2 = f_vals.copy()
    # Perturb b2
    W2_b2 = W2.copy()
    W2_b2[0, 0] += stepsize
    # Calculate y and f with the perturbation
    y_b2, _, _ = forward_propagation(X, W1, W2_b2, N, m)
    _, y_1st_b2, _ = compute_loss(X, W1, W2_b2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
    f_b2 = np.array([f(X[1, i], y_b2[0, i], y_1st_b2[0, i]) for i in range(1, N)])
    matrix_b2 -= f_b2
    # df/db2
    f_b2 = matrix_b2 @ residual
    # Weights 2
    matrix_w2 = np.zeros((m, N - 1)) + f_vals
    for j in range(m):  # For each weight
        # Perturb it
        W2_w2 = W2.copy()
        W2_w2[0, j + 1] += stepsize
        # Calculate y and f with the perturbation
        y_w2, _, _ = forward_propagation(X, W1, W2_w2, N, m)
        _, y_1st_w2, _ = compute_loss(X, W1, W2_w2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
        f_w2 = np.array([f(X[1, i], y_w2[0, i], y_1st_w2[0, i]) for i in range(1, N)])
        # Obtain the necessary array
        matrix_w2[j, :] -= f_w2
    # df/dw2
    f_w2 = (matrix_w2 @ residual).T
    # Then df/dW2
    dfdW2 = np.hstack((f_b2, f_w2[0]))

    ## Obtain dL/dW2
    # Inner terms
    dws = (2 / stepsize ** 2) * ((A_plus[:, 1:N] - 2 * A[:, 1:N] + A_minus[:, 1:N]) @ residual).T
    dW2 = np.insert(dws, 0, 0).reshape((1, m + 1))
    dW2 += (2 / stepsize) * dfdW2
    # Boundary terms
    if typea == 1:
        bt_w2_a = 2 * gamma * (y_hat[0, 0] - ya) * A[:, 0]
        bt_b2_a = 2 * gamma * (y_hat[0, 0] - ya)
    elif typea == 2:
```

```
        bt_w2_a = (2 * gamma / stepsize) * (y_1st[0, 0] - ya) * (A[:, 0] - A_minus[:, 0])
        bt_b2_a = 0

    if typeb == 1:
        bt_w2_b = 2 * gamma * (y_hat[0, -1] - yb) * A[:, -1]
        bt_b2_b = 2 * gamma * (y_hat[0, -1] - yb)
    elif typeb == 2:
        bt_w2_b = (2 * gamma / stepsize) * (y_1st[0, -1] - yb) * (A[:, -1] - A_minus[:, -1])
        bt_b2_b = 0

    bt_w2 = bt_w2_a + bt_w2_b
    bt_b2 = bt_b2_a + bt_b2_b
    # Total derivatives dL/dW2
    dW2[0, 1:] += bt_w2
    dW2[0, 0] += bt_b2

    return dW1, dW2


def train(X, f, W1, W2, tol, N, m, ya, typea, yb, typeb, stepsize, gamma):
    k = 1  # Iteration number
    # Compute loss gradients
    dW1, dW2 = back_propagation(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
    # Early stopping count
    wait = 0
    # Back to normal
    back = 0
    # Early stopping patience
    patience = 1000

    # Initialise loss (temp. variable)
    lossnow = tol + 1

    # Iterate until all components of the gradient are <= tol, or loss is <= tol
    while ((np.linalg.norm(dW1) > tol) | (np.linalg.norm(dW2) > tol)) & (lossnow > tol):
        # Obtain the gradients of loss function
        dW1, dW2 = back_propagation(X, W1, W2, N, m, stepsize, f, ya
```

```
, typea, yb, typeb, gamma)
        grad = np.concatenate((dW1.flatten(), dW2.flatten()))
        norm = np.linalg.norm(grad)

        # Define descent direction
        s1 = -dW1
        s2 = -dW2

        # Obtain an appropriate stepsize (backtracking Armijo)
        tau = np.random.rand() * 3/5  # number in (0,1) scaling down stepsize
        beta = np.random.rand() # bArmijo parameter
        a = 1  # Stepsize along negative gradient direction

        # Compute current loss and loss assuming stepsize a
        lossnow, _, _ = compute_loss(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
        lossnext, _, _ = compute_loss(X, W1 - a * dW1, W2 - a * dW2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
        while lossnext > (lossnow - beta * a * (norm ** 2)):
            # Decrease a until sufficient decrease of loss is achieved
            a *= tau
            # Compute new losses (current and for new a)
            lossnext, _, _ = compute_loss(X, W1 - a * dW1, W2 - a * dW2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)
            lossnow, _, _ = compute_loss(X, W1, W2, N, m, stepsize, f, ya, typea, yb, typeb, gamma)

        # Obtain the parameter values for the next iteration
        W1 += a * s1
        W2 += a * s2

        # Print loss
        if (k % 50 == 0):
            print(f'Iteration {k}\t Loss: {lossnow:.9f}\n'
                  f'dW1={np.linalg.norm(dW1)}\ndW2={np.linalg.norm(dW2)}')

        # Early stopping mechanism
        if (lossnow - lossnext) < 5e-5:
            wait += 1
            back = 1
```

```
                #print(f'Loss is not decreasing (for the {wait} time)')
                if wait >= patience:
                    print(f'Optimisation STOPPED at iteration {k}')
                    break
        else:
            back = 0

        if back == 0:
            wait = 0

        # Next iteration
        k += 1
    print(f'{k} iterations to reach a loss of: {lossnow}')

    return W1, W2, lossnow


import matplotlib.pyplot as plt
from scipy.integrate import solve_bvp
import time

# PLOTTING SETTINGS
plt.rc('text', usetex=True)
plt.rc('font', family='arial')
plt.rcParams.update({'font.size': 14})


def train_func(N, xlims, m, bc_type, bcs, f, tol=5e-3, eps=1e-3,
    gamma=10):
    a, b = xlims[0], xlims[1]
    y_a, y_b = bcs[0], bcs[1]

    # Create input matrix X
    x_vals = np.linspace(a, b, N+1)
    X = np.zeros((2, N+1))
    X[0, :] = np.ones(N+1)
    X[1, :] = x_vals

    # Initialise random weights and biases
    W1 = np.random.rand(m, 2)
    W2 = np.random.rand(1, m+1)
    W2[0, 0] = 0
```

```
    # Train the net
    start_time = time.time()
    W1, W2, loss = nn.train(X, f, W1, W2, tol, N, m, y_a, bc_type[0], y_b, bc_type[1], eps, gamma)
    end_time = time.time()
    training_time = end_time - start_time

    # Forward pass
    y_hat, _, _ = nn.forward_propagation(X, W1, W2, N, m)

    # Stats information
    print(f'Overall performance\n\tFinal loss: {loss}')
    print(f'\tTraining time: {training_time}')

    # Analytical sol.
    def sys(x, y):
        u1, u2 = y
        f_vals = f(x, u1, u2)
        return np.vstack((u2, f_vals))

    def bc(ya, yb):
        if bc_type[0] == 1:
            bc_a = ya[0]
        elif bc_type[0] == 2:
            bc_a = ya[1]
        elif bc_type[0] == 3:
            bc_a = ya[0] + ya[1]

        if bc_type[1] == 1:
            bc_b = yb[0]
        elif bc_type[1] == 2:
            bc_b = yb[1]
        elif bc_type[1] == 3:
            bc_b = yb[0] + yb[1]

        return np.array([bc_a - y_a, bc_b - y_b])

    y_initial_guess = np.zeros((2, x_vals.size))
    sol_exact = solve_bvp(sys, bc, x_vals, y_initial_guess)

    return x_vals, y_hat, sol_exact
```

```
if __name__ == '__main__':
    # Network parameters
    m = 40  # Number of neurons in hidden layer

    # Training parameters
    N = 40  # Number of inputs
    gamma = 10
    tol = 1e-2
    eps = 1e-3  # Perturbation for finite differences

    # Domain
    xlims = [0, np.pi]
    bc_types = [(1,1), (1,2), (2,1), (2,2)]
    bcs = [0, 1]


    # State the problem & boundary conditions
    def f(x, y, y_1st):
        # Returns function
        return -2*x**2 + y


    xs = {}
    y_hats = {}
    ys = {}

    for idx, bc_type in enumerate(bc_types):
        x, y_hat, sol = train_func(N, xlims, m, bc_type, bcs, f, tol
=1e-2)
        xs[idx] = x
        y_hats[idx] = y_hat
        ys[idx] = sol.sol(x)[0]

    fig, a = plt.subplots(2, 2, figsize=(10,6))
    plt.subplots_adjust(top=0.8, wspace=0.1)

    a[0, 0].scatter(xs[0], y_hats[0], label='Network predictions $\
hat{y}$', color='orange', s=10, zorder=2)
    a[0, 0].plot(xs[0], ys[0], label='Numerical solution', color='
blue', linewidth=2, zorder=1)
    a[0, 0].set_title('Dirichlet boundary conditions')
    a[0, 0].set_ylabel('$y$')
```

```
    plt.setp(a[0, 0].get_xticklabels(), visible=False, fontsize=12)
    plt.setp(a[0, 0].get_yticklabels(), fontsize=12)

    a[0, 1].scatter(xs[1], y_hats[1], color='orange', s=10, zorder
=2)
    a[0, 1].plot(xs[1], ys[1], color='blue', linewidth=2, zorder=1)
    a[0, 1].set_title('Dirichlet-Von Neumann boundary conditions')
    plt.setp(a[0, 1].get_xticklabels(), visible=False, fontsize=12)
    plt.setp(a[0, 1].get_yticklabels(), fontsize=12)

    a[1, 0].scatter(xs[2], y_hats[2], color='orange', s=10, zorder
=2)
    a[1, 0].plot(xs[2], ys[2], color='blue', linewidth=2, zorder=1)
    a[1, 0].set_title('Von Neumann-Dirichlet boundary conditions')
    a[1, 0].set_ylabel('$y$')
    a[1, 0].set_xlabel('$x$')
    plt.setp(a[1, 0].get_xticklabels(), fontsize=12)
    plt.setp(a[1, 0].get_yticklabels(), fontsize=12)

    a[1, 1].scatter(xs[3], y_hats[3], color='orange', s=10, zorder
=2)
    a[1, 1].plot(xs[3], ys[3], color='blue', linewidth=2, zorder=1)
    a[1, 1].set_title('Von Neumann boundary conditions')
    a[1, 1].set_xlabel('$x$')
    plt.setp(a[1, 1].get_xticklabels(), fontsize=12)
    plt.setp(a[1, 1].get_yticklabels(), fontsize=12)

    fig.legend(loc='upper center')
    plt.tight_layout()
    plt.show()
```