Solving Partial Differential Equations With Neural Networks

Candidate Number: 1068160

Abstract

In this project, we proposed a neural network approach to solve ODEs and PDEs. We discussed how the derivatives may be evaluted from the neural network and how the loss function is designed to ensure the output from the neural network satisfies the PDE constraints and the boundary and initial conditions. We also tested different optimisation algorithm and the performance of the neural network on problems with multiple solutions.

1 Introduction

Many methods have been developed for solving ordinary differential equations (ODEs) and partial differential equations (PDEs). Most of the methods involve discretisation of the domain. For example, the finite difference method (FDM) approximates the derivatives in PDEs using discrete differences between neighbouring points on a grid, and the finite element method (FEM) divides the domain into smaller elements and represents the solution as a linear combination of basis functions. In this case study, we proposed a grid-free method of solving ODEs and PDEs using artificial neural networks (ANNs).

The method was first proposed in the paper [1] in 1994. The idea takes advantage of the approximate capabilities of feed-forward neural networks and is thus able to construct a continuous and differentiable solution to the PDEs. During the past decade, we have achieved advances in deep learning software tools, especially Auto-grad implementation and optimisation algorithms, as well as hardware developments such as GPUs and TPUs.

Those developments allow us to extend the structure in the paper [1] to deep neural networks. The Auto-grad enables the rapid computation of the gradient of the error with respect to the network parameters, as well as the computation of the partial derivatives from the neural networks.

Utilizing a neural network architecture brings several appealing features to the method:

1. The solution from ANNs is a differentiable, closed analytic form that can be easily used in subsequent calculations. In contrast, most other techniques provide a discrete solution or a solution with limited differentiability.

2. The neural network approach does not rely on grids or meshes and can handle complex geometries and boundary conditions more naturally.

3. This approach can be built upon deep learning frameworks, such as TensorFlow and PyTorch, which means they can take advantage of the latest advancements in optimisation algorithms, hardware accelerators (e.g., GPUs), and parallel computing techniques to improve the efficiency and accuracy of solving PDEs.

4. The neural network can be implemented naturally with Auto-grad which is fast and does not the numerical instability compared to methods like finite difference.

2 Description of the method

2.1 Problem setup

Consider the second-order PDE problem with solution $u(\mathbf{x})$ on the domain $\Omega \subset \mathbb{R}^d$ with $\mathbf{x} = (x_1, \ldots, x_d)$:

$$f(\mathbf{x}, u) = f\left(\mathbf{x}; \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_d}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_d}\right) = 0, \quad \mathbf{x} \in \Omega \tag{1}$$

With boundary condition:

$$\mathcal{B}(u, \mathbf{x}) = 0, \quad \text{on} \quad \partial\Omega$$

If we have $x_1 = t$ standing for time, we usually also have the initial condition:

$$\mathcal{I}(u, \mathbf{x}) = 0, \quad \text{on } x_1 = 0$$

For Dirichlet boundary conditions, we have:

$$u(\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \tag{2}$$

For von Neumann boundary conditions, we have:

$$\frac{\partial u}{\partial n}(\mathbf{x}) = h(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega \tag{3}$$

By the universal approximation theorem of the neural network [2], the above problem can be approximated by a neural network. The residuals of the PDEs are encoded in the neural network loss function as constraints, ensuring that the network's outputs satisfy the PDEs, initial conditions, and boundary conditions.

2.2 The neural network structure for PDE

We define D layers of fully connected feed-forward neural networks:

$$N := L_D \circ \sigma \circ L_{D-1} \circ \sigma \circ \cdots \circ \sigma \circ L_1$$

where:

$$\begin{aligned} L_1(\mathbf{x}) &:= W^{(1)}\mathbf{x} + b^{(1)}, \quad W^{(1)} \in \mathbb{R}^{d_1 \times d}, b_1 \in \mathbb{R}^{d_1} \\ L_i(\mathbf{x}) &:= W^{(i)}\mathbf{x} + b^{(i)}, \quad W^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}, b_i \in \mathbb{R}^{d_i}, \forall i = 2, 3, \cdots D-1 \\ L_D(\mathbf{x}) &:= W^{(D)}\mathbf{x} + b^{(D)}, \quad W^{(D)} \in \mathbb{R}^{1 \times d_{D-1}}, b_D \in \mathbb{R}^1 \end{aligned}$$

where $W^{(i)}$ is the weight matrix on ith layer and b_i is the bias vector on ith layer. σ is the activation function which we should define. We denote $N(\mathbf{x})$ as the output from the neural network and $u(\mathbf{x})$ is approximated by $N(\mathbf{x})$. We define the partial differential equation loss (PDE loss) as follows:

$$\mathcal{L}_{PDE} = |f(\mathbf{x}, N)|^2 = \left| f\left(\mathbf{x}; \frac{\partial N}{\partial x_1}, \ldots, \frac{\partial N}{\partial x_d}; \frac{\partial^2 N}{\partial x_1 \partial x_1}, \frac{\partial^2 N}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^2 N}{\partial x_d \partial x_d}\right)\right|^2, \quad (4)$$

Similarly, we define the boundary condition loss (BC loss) and initial condition loss(IC loss):

$$\mathcal{L}_{BC} = |\mathcal{B}(N, \mathbf{x})|^2, \quad (5)$$

$$\mathcal{L}_{IC} = |\mathcal{I}(N, \mathbf{x})|^2, \quad (6)$$

And the total loss is defined as a weighted sum of PDE loss, BC loss and IC loss:

$$\mathcal{L} = \frac{1}{w}(\mathcal{L}_{PDE} + \lambda_{IC}\mathcal{L}_{IC} + \lambda_{BC}\mathcal{L}_{BC}), \quad (7)$$

where λ_{IC} and λ_{BC} are the weighting parameters for IC loss and BC loss and w is the normalisation factor.

2.3 The back-propagation and Auto-grad

The training of the neural network requires evaluation of the derivatives of the loss function with respect to the weighting parameters and bias at each layer, i.e $\frac{\partial \mathcal{L}}{\partial W_{i,j}^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b_i^{(l)}}$. In addition, the loss function includes the derivative of the neural output N with respect to the input vector $\mathbf{x}$: $\frac{\partial N(\mathbf{x})}{\partial x_i}$, which also needs to be evaluated. Here, we give an example of evaluating the derivatives explicitly with a one-hidden layer.

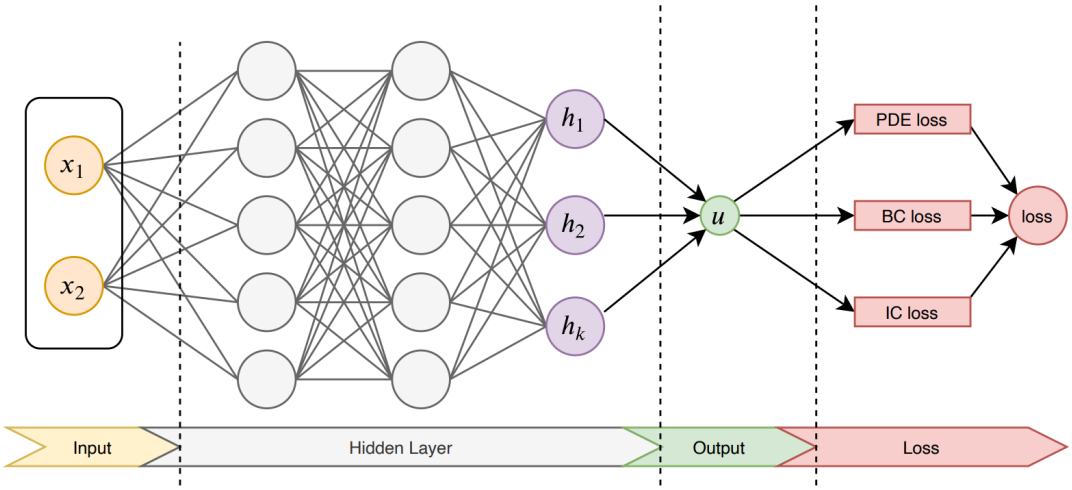


Figure 1: illustration of neural network for PDEs[3]

Denote the input vector as $\mathbf{x} = (x_1, x_2, \ldots, x_d)^T$, and the structure of the feed-forward neural network can be written as:

$$N := L_2 \circ \sigma \circ L_1$$

$$L_1(\mathbf{x}) := W^{(1)}\mathbf{x} + b^{(1)}, \quad W^{(1)} \in \mathbb{R}^{N\times d}, b^{(1)} \in \mathbb{R}^N$$
$$L_2(\mathbf{x}) := W^{(2)}\mathbf{x} + b^{(2)}, \quad W^{(2)} \in \mathbb{R}^{1\times N}, b^{(2)} \in \mathbb{R}^1$$

$$N(\mathbf{x}) = \sum_{i=1}^{N} W_{1,i}^{(2)}\sigma(z_i) + b^{(2)} \text{ with } z_i = \sum_{j=1}^{d} W_{i,j}^{(1)}x_j + b_i^{(1)} \tag{8}$$

The derivatives can be evaluated through the chain rule:

$$\frac{\partial^k N(\mathbf{x})}{\partial x_j^k} = \sum_{i=1}^{N} W_{1,i}^{(2)} \left(W_{i,j}^{(1)}\right)^k \sigma_i^{(k)} \tag{9}$$

where $\sigma_i = \sigma(z_i)$ and $\sigma^{(k)}$ denotes the k^{th} order derivative of the sigmoid function. Generally, the gradient with this neural network is given by:

$$N_g(\mathbf{x}) = \frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_1}}\frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}}\cdots\frac{\partial^{\lambda_n}}{\partial x_n^{\lambda_n}}N = \sum_{i=1}^{n} W_{1,i}^{(2)} P_i \sigma_i^{(\Lambda)}$$
$$P_i = \prod_{k=1}^{n} (W_{i,k}^{(1)})^{\lambda_k} \tag{10}$$

The derivatives of neural networks with more hidden layers can be evaluated similarly. In this project, the derivatives are evaluated through the Auto-grad function

of PyTorch. Regarding the training of the neural network, we need to evaluate the derivatives of the loss function with respect to the network parameters. The gradient of the loss function with respect to the weights $W^{(i)}$ and biases $b^{(i)}$ for each layer is calculated using the chain rule. As the $\mathcal{L}_{PDE}$ involves gradients of the neural network N_g, we also need to evaluate the gradient of N_g with respect to the network parameters.

$$\frac{\partial \mathcal{L}(N, N_g)}{\partial W_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(N, N_g)}{\partial N} \frac{\partial N}{\partial W_{i,j}^{(l)}} + \frac{\partial \mathcal{L}(N, N_g)}{\partial N_g} \frac{\partial N_g}{\partial W_{i,j}^{(l)}} \tag{11}$$

With the neural network structure with one hidden layer, it can be shown the derivatives of N_g with respect to the network parameters:

$$\begin{aligned} \frac{\partial N_g}{\partial W_{1,i}^{(2)}} &= P_i \sigma_i^{(\Lambda)}, \\ \frac{\partial N_g}{\partial b_i^{(1)}} &= W_{1,i}^{(2)} P_i \sigma_i^{(\Lambda+1)}, \\ \frac{\partial N_g}{\partial W_{i,j}^{(1)}} &= x_j W_{1,i}^{(2)} P_i \sigma_i^{(\Lambda+1)} + W_{1,i}^{(2)} \lambda_j (W_{i,j}^{(1)})^{\lambda_j - 1} \left(\prod_{k=1, k \neq j} (W_{i,k}^{(1)})^{\lambda_k} \right) \sigma_i^{(\Lambda)}, \end{aligned} \tag{12}$$

and the derivatives of N with respect to the network parameters are given by:

$$\begin{aligned} \frac{\partial N}{\partial W_{1,i}^{(2)}} &= \sigma_i, \\ \frac{\partial N}{\partial b_i^{(1)}} &= W_{1,i}^{(2)} \sigma_i^{(1)}, \\ \frac{\partial N}{\partial W_{i,j}^{(1)}} &= x_j W_{1,i}^{(2)} \sigma_i^{(1)}, \end{aligned} \tag{13}$$

Once the gradients have been computed, they are used to update the weights and biases during the optimisation process. We use gradient descent to optimise the weights and bias:

$$W_{i,j}^{(l)} = W_{i,j}^{(l)} - r \frac{\partial \mathcal{L}}{\partial W_{i,j}^{(l)}} \tag{14}$$

$$b_i^{(l)} = b_i^{(l)} - r \frac{\partial \mathcal{L}}{\partial b_i^{(l)}} \tag{15}$$

where r is the learning rate.

3 Numerical Experiments

3.1 Second order ordinary differential equation

We applied the neural network to solve:

$$y'' = f(x)$$

with boundary conditions:

$$y(0) = 1, y(1) = 1$$

in the interval:

$$x \in [0, 1]$$

Regarding the training set, we chose 100 equally spaced points on $[0, 1]$. We have 2 points at the boundary $x = 0$ and $x = 1$ with 98 points inside the interval with $\Delta x = \frac{1}{98}$. We proposed a fully connected feed-forward neural network with one hidden layer with 6 neurons. As we are solving on a single dimension, our input is a 1-D scalar: $\mathbf{x} = x$, and the structure of the feed-forward neural network can be written as:

$$N := L_2 \circ \sigma \circ L_1$$
$$L_1(\mathbf{x}) := W^{(1)}\mathbf{x} + b^{(1)}, \quad W^{(1)} \in \mathbb{R}^{6\times 1}, b^{(1)} \in \mathbb{R}^6$$
$$L_2(\mathbf{x}) := W^{(2)}\mathbf{x} + b^{(2)}, \quad W^{(2)} \in \mathbb{R}^{1\times 6}, b^{(2)} \in \mathbb{R}^1$$

We used the following loss function to train our model:

$$\mathcal{L} = \frac{1}{98 + 2\lambda}\left(\sum_{j=1}^{98}(\widehat{y''}(x_j) - f(x_j))^2 + \lambda(\widehat{y(1)} - y(1))^2 + \lambda(\widehat{y(0)} - y(0))^2\right) \tag{16}$$

where $\widehat{y''}, \widehat{y}$ were outputs from the neural network and the derivatives are evaluated using the formula 9. We set $\lambda = 1000$ and performed the training to solve the problem where $f(x) = -2$. In this case, the exact solution is $y_{sol}(x) = 1 + x(1 - x)$. We used full-batch gradient descent to optimise the algorithm. The full-batch gradient descent means at each step, we calculated the loss function using all the training samples, which were the 100 x grid points in our case. Then we updated the weighting parameters and bias in our neural network with a learning rate $r = 0.01$.

Figure 2 shows our training results. We trained the model on grids of 100 points. After 1000 epochs of training, we reached about 0.001 in error across the interval. Note that the error plot in Figure 2 is plotted with 1000 equally spaced points in the interval $[0, 1]$ where our training used 100 equally spaced points. We achieved high-accuracy solutions not only on the points which are included in our training set but also on the points which are not.

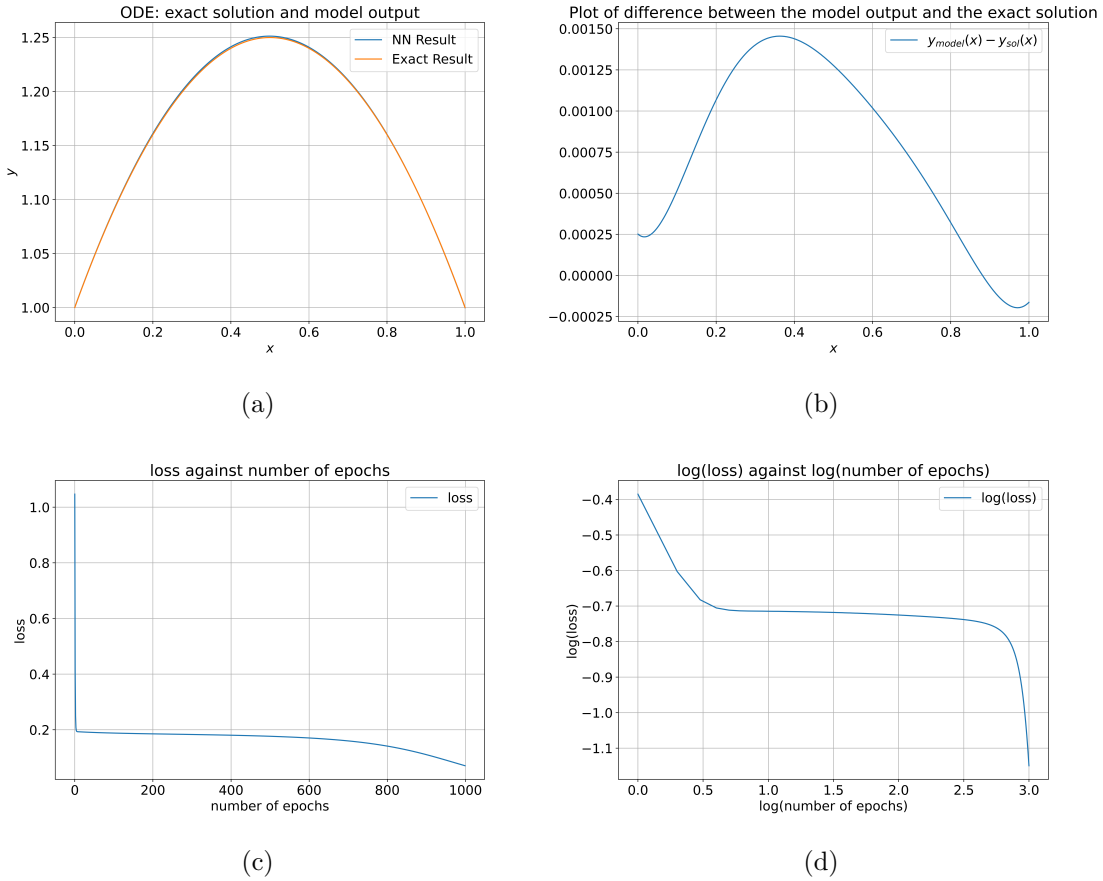


Figure 2: (a) Output of NN and exact solution (b) Error of NN model (c) Loss (d) log(loss)

3.2 Second order partial differential equation

We tried to apply the neural network to solve the following 2nd-order PDE:

$$\Delta\Psi(x,y) = f(x,y) = (2 - \pi^2 y^2) sin(\pi x) \tag{17}$$

on $0 \leq x \leq 1$, $0 \leq y \leq 1$, where $\Delta\Psi(x,y) = \frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2}$ with a mixed Dirichlet and von-Neumann boundary conditions: $\Psi(0,y) = 0$, $\Psi(1,y) = 0$, $\Psi(x,0) = 0$, $\frac{\partial}{\partial y}\Psi(x,1) = g(x) = 2sin(\pi x)$. The exact solution is $\Psi(x,y) = y^2 sin(\pi x)$.
We proposed a feed-forward neural network with two hidden layers with 128 neurons with the input vector $\mathbf{x} = (x,y)^T$:

$$N := L_3 \circ \sigma \circ L_2 \circ \sigma \circ L_1$$

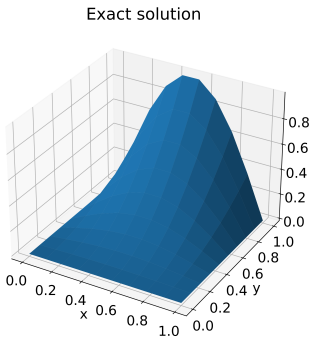


Figure 3: The plot of the exact solution to the PDE problem

$$
\begin{aligned}
L_1(\mathbf{x}) &:= W^{(1)}\mathbf{x} + b^{(1)}, \quad W^{(1)} \in \mathbb{R}^{128\times 1}, b^{(1)} \in \mathbb{R}^{128} \\
L_2(\mathbf{x}) &:= W^{(2)}\mathbf{x} + b^{(2)}, \quad W^{(2)} \in \mathbb{R}^{128\times 128}, b^{(2)} \in \mathbb{R}^{128} \\
L_3(\mathbf{x}) &:= W^{(3)}\mathbf{x} + b^{(3)}, \quad W^{(3)} \in \mathbb{R}^{1\times 128}, b^{(3)} \in \mathbb{R}^{1}
\end{aligned}
$$

We discretise our region of $(x, y) \in [0, 1] \times [0, 1]$ to a 10×10 mesh grids where points are equally spaced:

$$(x_i, y_j) \text{ where } x_i = i\Delta x, y_j = j\Delta y, \quad i, j \in \{0, 1, 2, \ldots, 9\}$$

Here, $\Delta x = \Delta y = \frac{1}{9}$ is the distance between neighbouring grid points, and we use i and j as indices to refer to the x and y coordinates, respectively. The set notation $0, 1, 2, \ldots, 9$ indicates that i and j take on values in the range from 0 to 9, inclusive. Figure 4 gives the plot of the mesh grid. We have 26 points on the boundary where we enforced the boundary conditions. We have 8×8 points inside. For inside points, we have a relative weight of 1 for the PDE loss, and for outside (boundary) points, we have a weight of λ for the BC loss.

Our loss function can be expressed as:

$$
\begin{aligned}
\mathcal{L} = \frac{1}{64 + 26\lambda} \Bigg[& \sum_{i,j\in\{1,2,\ldots,8\}} \left(\frac{\partial^2 \widehat{\Psi}(x_i, y_j)}{\partial x^2} + \frac{\partial^2 \widehat{\Psi}(x_i, y_j)}{\partial y^2} - f(x_i, y_j) \right)^2 \\
& + \lambda \sum_{i=0,j\in\{0,1,\ldots,9\}} \left(\widehat{\Psi}(x_i, y_j) - 0 \right)^2 + \lambda \sum_{i=9,j\in\{0,1,\ldots,9\}} \left(\widehat{\Psi}(x_i, y_j) - 0 \right)^2 \\
& + \lambda \sum_{i=\{1,2,\ldots,8\},j=0} \left(\widehat{\Psi}(x_i, y_j) - 0 \right)^2 + \lambda \sum_{i=\{1,2,\ldots,8\},j=9} \left(\frac{\partial \widehat{\Psi}(x_i, y_j)}{\partial y} - g(x_i) \right)^2 \Bigg]
\end{aligned}
$$

where $\Psi(x, y)$ is the output of the neural network and the derivatives are evaluated using the Auto-grad algorithm. Figure 5 shows our training results. The model was

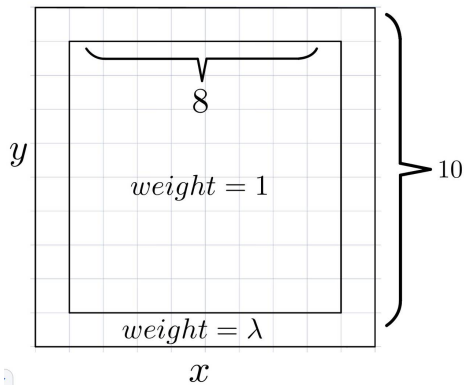


Figure 4: Schematic plot of the 10×10 mesh grid

trained using the gradient descent algorithm with learning rate $r = 0.01$. The effect of the choice of the optimisation algorithm will be discussed in the next section. After 1000 epochs of training, we have reached an average error of about 0.002 over the domain.

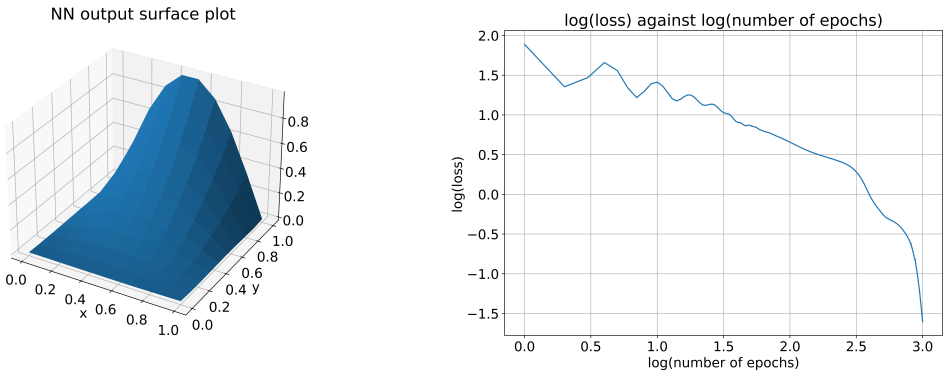
4 Further Developments (Personal Extensions)

4.1 Problems with non-unique solutions

Suppose we use the neural network to fit:

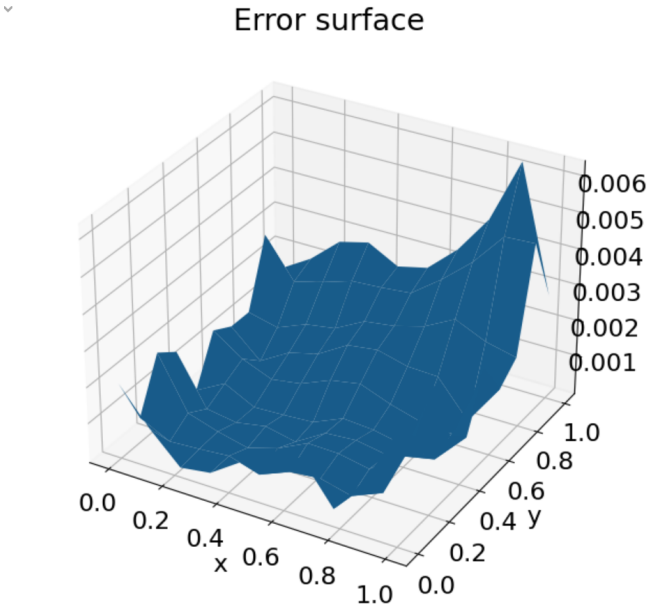
$$y^2 = x \tag{18}$$

on $[0, 1]$. We used 200 training points, 100 from the upper branch and 100 from the lower branch, as shown in Figure 6 (a). We applied the same neural network structure as the one we used for solving the second-order ODE problem, which had one hidden layer with 6 neurons. During the training process, the output of the NN model oscillated between the upper branch (mode 1) and lower branch (mode 2) and ultimately converged to one of the branches. Thus, the trained model would give an output of one branch of $y^2 = x$, and which branch the model would converge to was random. Randomness was introduced in our initialisation of the neural network. In this project, we chose to initialise the weights with a random value sampled from a normal distribution with mean 0 and standard deviation 1. Different initial values of the parameters of the neural network lead to different results of convergence.



(a)

(b)

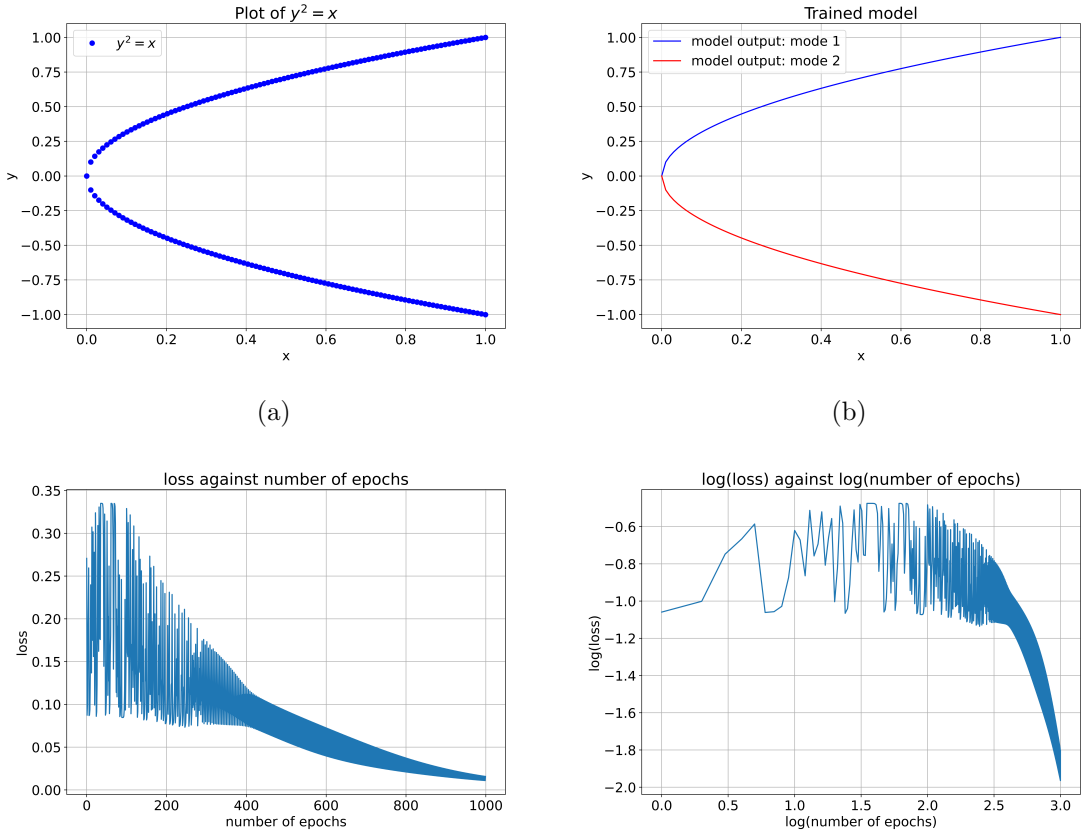


(c)

Figure 5: (a) Output Surface of the trained NN model (b) log(loss) against log(epochs) (c) Difference between the NN output surface and the exact solution

4.2 Optimisation algorithm

In previous examples, we used full-batch gradient descent. Full-batch here stands for calculating the loss over all the training points before we perform gradient descent on the parameters. We introduced the mini-batch gradient descent, where we divided the training points into small sets (batches), and for each epoch, we calculated the loss on each small batch and perform gradient descent. In the example of the ODE



(a) (b)

(c) (d)

Figure 6: (a) The training points (b) Output of the NN which has two modes (c) loss against the number of epochs trained (d) log(loss) against the number of epochs trained. The oscillatory pattern in the loss function is due to the model output is jumping between the upper branch (mode 1) and lower branch (mode 2)

problem in the previous section, we divided the loss function into:

$$\mathcal{L}_{left} = \frac{1}{n+\lambda-1}(\sum_{j=1}^{n-1}(\widehat{y''}(x_j) - f(x_j))^2 + \lambda(\widehat{y(0)} - y(0))^2)$$

$$\mathcal{L}_{interval} = \frac{1}{n}(\sum_{j=k}^{k+n-1}(\widehat{y''}(x_j) - f(x_j))^2)$$

$$\mathcal{L}_{right} = \frac{1}{n+\lambda-1}(\sum_{j=N-n+1}^{N}(\widehat{y''}(x_j) - f(x_j))^2 + \lambda(\widehat{y(1)} - y(1))^2)$$

where n was the batch size and N was the total number of training points. The training points were divided into batches with n points except the batches containing

the endpoints, where we had a boundary point with weight λ on the loss contribution. We used the exact same neural network structure and the same learning rate as in the ODE example in the previous section. We used $n = 5$ so the 100 training points were divided into 20 batches. In each epoch, we calculated the loss of each batch and perform gradient descent 20 times. To compare full-batch gradient descent and mini-batch gradient descent, we introduced the mean squared error (MSE):

$$e^2 = \frac{1}{100} \sum_{i=0}^{i=99} \left(\widehat{y}(x_i) - y_{sol}(x_i)\right)^2 \tag{19}$$

where $\widehat{y}(x)$ was the model output and y_{sol} was the exact solution of the ODE problem.

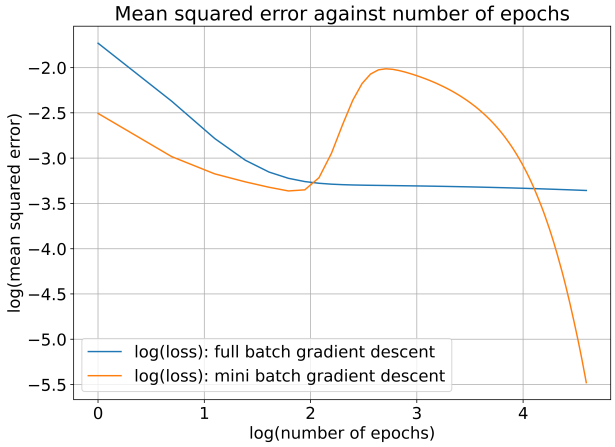


Figure 7: log(MSE) against log(number of epochs) with two different batch sizes

Figure 7 gives the plot of the mean squared error of the two different optimisation algorithms. As we can see, with all other parameters being the same, the mini-batch gradient descent performs better as we reach lower MSE with the same number of epochs trained.

5 Conclusions

In this project, we introduced a neural network approach to solve PDEs and ODEs. We discussed theoretically how the derivatives were evaluated from the neural network as well as the construction of the loss function to enforce the constraints of the problem. We tested the algorithm on one second-order ODE problem and one second-order PDE problem and reached high accuracy. We improved the convergence

rate by dividing the training data into small batches and performing gradient descent on each batch.

We also tested the performance of the algorithm on a problem with multiple solutions. We found that during the training process, the neural network output jumped between the possible solutions and finally converge on one of them.

In this project, we only considered the constraints of the PDEs. The loss function could be extended to consider real data observations as well, which is now called "Physics-Informed Neural Networks" (PINN). The PINN is now applied to many complex problems for its ability to tackle high-dimensional problems. Further development of the project may involve a new design of the loss function to balance the PDE constraints and observed data. We may also try to GPUs to accelerate our training process.

References

[1] MWMG Dissanayake and Nhan Phan-Thien. Neural-network-based approximations for solving partial differential equations. *Communications in Numerical Methods in Engineering*, 10(3):195–201, 1994.

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

[3] Shuhong Wang, Meng Chen, Maziar Raissi, and Paris Perdikaris. Accelerating physics-informed neural network training with prior dictionaries. *arXiv preprint arXiv:2004.08151*, 2020.